MINIMAL DEPLOYABLE ENDPOINT-DRIVEN NETWORK FORWARDING: PRINCIPLE, DESIGNS AND APPLICATIONS

BY

ZHUOTAO LIU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Associate Professor Yih-Chun Hu, Chair
Professor Klara Nahrstedt
Assistant Professor Prateek Mittal, Princeton University
Associate Professor Michael Bailey

ABSTRACT

Networked systems now have significant impact on human lives: the Internet, connecting the world globally, is the foundation of our information age, the datacenters, running hundreds of thousands of servers, drive the era of cloud computing, and even the Tor project, a networked system providing online anonymity, now serves millions of daily users.

Guided by the end-to-end principle, many computer networks have been designed with a simple and flexible core offering general data transfer service, whereas the bulk of the application-level functionalities have been implemented on endpoints that are attached to the edge of the network. Although the end-to-end design principle gives these networked systems tremendous success, a number of new requirements have emerged for computer networks and their running applications, including untrustworthy of endpoints, privacy requirement of endpoints, more demanding applications, the rise of third-party Intermediaries and the asymmetric capability of endpoints and so on. These emerging requirements have created various challenges in different networked systems.

To address these challenges, there are no obvious solutions without adding in-network functions to the network core. However, no design principle has ever been proposed for guiding the implementation of in-network functions. In this thesis, We propose the first such principle and apply this principle to propose four designs in three different networked systems to address four separate challenges. We demonstrate through detailed implementation and extensive evaluations that the proposed principle can live in harmony with the end-to-end principle, and a combination of the two principle offers more complete, effective and accurate guides for innovating the modern computer networks and their applications.

To my parents and my dear wife.

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my thesis advisor, Professor Yih-Chun Hu, for his invaluable guidance and continuous support over the past five years. I am so grateful to him for his persistent encouragement regardless of the ups and downs during my Ph.D. study. It is so inspiring to work with him, one of the smartest persons that I have ever met.

Next, I would like to sincerely thank my other committee members, Professor Klara Nahrstedt, Professor Prateek Mittal and Professor Michael Bailey. Their constructive suggestions and comments helped me significantly improve my thesis. I am truly honored to have them on my doctoral committee.

Further, I would like to sincerely thank Professor Kai Chen for this great support during my academic visit at Hong Kong University of Science and Technology. Thank Dr. Qiang Xu for his tremendous support during my academic internship at NEC Labs, America.

I would like to extend my gratitude to the professors, researchers, and colleagues who helped me on this thesis and other research and course projects. In particular, I would like to thank Professor William Sanders, Professor Carl Gunter, Professor Brighten Godfrey, Professor Robin Kravets, Professor Julia Hockenmaier, Professor Indranil Gupta, Professor Sang-Yoon Chang, Hao Jin, Dr. Haitao Wu, Dr. Yi Wang, Dr. Haiming Jin, Dr. Wei Bai, Shuihai Hu, Hong Zhang and Cheng Jin for valuable discussions, suggestions, and collaborations.

I would also like to thank all of my friends who supported me during these years. I am so happy to have you guys to share my time.

Lastly, and most importantly, I wish to thank my family. Thank my father Jinguo Liu and my mother Yuqing Zhao for their priceless love ever since I was born. Thank my dearest wife Shasha Dai who chose to spend the rest of her life together with me. I am so fortunate to have her tremendous love and support. To them I dedicate this thesis.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Motivations and Challenges

The end-to-end principle articulates the fundamental principle of how the computer networks have been designed. The end-to-end principle concerns how application requirements should be implemented in a networked system. In particular, the principle suggests that application-level functions usually cannot, and preferably should not, be built into the core of the network, which was stated as follows in the original paper [1]: *The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the endpoints of the communications system. Therefore, providing that questioned function as a feature of the communications systems itself is not possible.* Many people attribute the tremendous success of the computer networking (*e.g.,* the Internet) to this end-to-end principle.

Guided by the end-to-end principle, over the past few decades, many computer networks have been designed with a simple and flexible core offering very general data transfer service, whereas the bulk of the application-level functionalities have been implemented on endpoints that are attached to the edge of the network. However, as our information age continues to thrive, a number of new requirements have emerged for computer networks and their running applications. These various new requirements create a set of new dimensions for designing computer networks. In this section, from the following five dimensions, we argue for the need of a new design principle to inter-operate with the classic end-to-end principle for computer network designs.

1.1.1 Untrustworthy of Endpoints

The end-to-end principle was first articulated assuming that all endpoints are trusted to obey application-level protocols for the wellness of entire system. However, in open computer networks like the Internet, given the tremendous growth in the number of devices connected to the network and dramatic diversification of motivations for using the network, there is no reason to believe that an endpoint will behave as desired. As a consequence, malicious or compromised endpoints are constructed to launch various abuse or attacks on a networked system as a whole to make the network less usable and attacks on a specific endpoint (*e.g.,* a competitor) to knock it offline for the best interest of an adversary. To make a network more trustworthy while without trusting endpoints, it seems inevitable that additional functionalities are demanded in the network core.

Perhaps the volumetric Distributed Denial of Service (DDoS) attacks are and will continue to be the most disruptive attacks since the invention of the Internet. Due to the connectionless of the network layer (designed based on the end-to-end principle), an Internet endhost can send any IP packets to any other endhosts without obtaining the receiver's permission. As a result, a target server cannot control which sources are sending traffic to it and how much traffic is delivered to it. As a consequence, a concerted effort from a group of malicious/compromised endpoints can easily overwhelm a target server's network with unwanted traffic, knocking off the target server offline. The end-to-end principle would say the target server is responsible for protecting itself from attacks via various control mechanisms implemented on the target server (*i.e.,* intrusion detection systems, anti-virus software and hardware), the volumetric DDoS attack traffic, however, must be handled *inside* the network and before all bottleneck network links.

Consider the Tor network as a second example. Tor is the most widely used anonymous network, which is an overlay built on a set of globally located Internet hosts, called relays. Although Tor is designed to be used by legitimate users to protect their online privacy, malicious Tor clients often abuse Tor as a stepping stone to launch various attacks against websites, including content scraping, comment spamming and vulnerability scanning. To give some sense, based on data from Project Honey Pot, 18% of global email spam, or approximately 6.5 trillion unwanted messages per year, begin with an automated bot

harvesting email addresses via Tor [2]. As a response, websites take radical countermeasures against Tor traffic, including completely block Tor or serving endless CAPTCHA challenges for Tor clients. Since all client requests, from both legitimate users and abusive endpoints, are multiplexed by Tor relays to protect their privacy, the above countermeasures that ideally should only be applied against abusive endpoints equally apply to legitimate users, which makes Tor less usable for these legitimate users. Making Tor more trustworthy, without breaking the privacy of Tor users, implies additional support and functionality from the center of the Tor network.

1.1.2 Privacy of Endpoints

In an era of mass surveillance, our online communications are being increasingly monitored by many third parities (*e.g.,* businesses and government entities) to infer sensitive information. However, there are a number of circumstances where a desire for online privacy might arise, including law enforcement, intelligence agencies, political dissidents, journalists, whistle-blowers, businesses and ordinary citizens looking for privacy enhancement. The desire for anonymity represents an extreme situation of un-aligned interest of the two endpoints: one endpoint may wish to hide its identity whereas the other endpoint may require such identity for policy-compliance control, such as assets protection, action authorization and so on.

Note that the requirement for endpoint privacy itself may not break the underlining assumption of the end-to-end principle since trusted anonymous endpoints can still behave as desired. However, together with the untrustworthy nature of endpoints (described in Section 1.1.1), endpoint privacy violates the assumption of end-to-end principle to a even larger extent: not only endpoints are untrusted, but also the network can not account their activities. Therefore, in a computer network that offers endpoint privacy, without in-network support, it is almost certainly that no desired goals can be achieved by protocols and algorithms that are merely implemented on endpoints.

The Tor network, again, is a perfect example here. There has been a long tension between the Tor network and many websites content delivery network (CDN) providers (*e.g.,* Cloudflare and Akamai) that these CDN providers often block Tor traffic as a whole or offer limited service for Tor requests, or serve nu-

merous challenges. This is neither because these providers intentionally discriminate Tor users nor these providers did not implement any anti-abuse defense mechanism. It is simply because there is no way out only based on the effort from these providers (endpoints), without having additional functionalities inside the Tor network.

1.1.3 More Demanding Applications

The end-to-end principle essentially proposes a "best-effort" service model of networked systems: the network makes no guarantee about the performance (*e.g.,* throughput, latency) that a particular application can achieve at any moment. As the rise of cloud computing, various business-critical applications have been deployed on multi-tenant datacenters. Although tenants are typically allocated with guaranteed compute and storage resource when provisioning virtual machines, the multi-tenant networks, guided by the end-to-end principle, are still at the phase of offering best-effort delivery services. Consequently, datacenter networks quickly become the bottleneck of high demanding applications in the cloud.

Big data analytic application is one of the most popular applications deployed in multi-tenant datacenters. Big data analytic workloads generated by the popular frameworks (such as Hadoop and Spark) can involve massive amounts of data exchange among network endpoints (*i.e.,* servers). For instance, the shuffle process bridging the Map phase and Reduce phase of MapReduce jobs can manipulate terabytes or even petabytes of data among tens of thousands of machines. To give some sense, Hadoop traces from Facebook show that, on average, transferring data between successive stages accounts for 33% of the running times of jobs with reduce phases [3]. In frameworks like MapReduce, a stage cannot complete before it receives all the data from its previous stage. As a result, a single "laggard" task can dramatically delay the entire job. Therefore, offering guaranteed bandwidth for tenant VMs becomes desirable to ensure predictable lower bound performance for tenant applications. Augmenting network with such capabilities, clearly, requires additional mechanisms in the network core.

Virtual Private Cloud (VPC) has seen a rapid growth in multi-tenant datacenters. The biggest upside of VPC is that tenants can optimize various perfor-

mance metrics for business-critical applications. Under the end-to-end principle, optimization effort should be made on endpoints (severs), including designing more efficient transport control protocol [4] for datacenters, improving computation models to reduce data transfers. However, for high-demanded applications, for instance requiring bounded latency [5, 6], optimization can never been done solely at endpoints if the underlining network connecting these endpoints offers only best-effort delivery.

1.1.4 The Rise of Third-Party Intermediaries

An undeniable fact of the Internet is the increasing third-party involvement in communications between network endpoints. These third parties include cloud service providers (*e.g.,* CDN providers for accelerating content delivery), government officials whose interests range from taxation to law enforcement and public safety, and cooperate organizations (*e.g.,* Internet Service Providers, ISPs) that enforce organizational policies. Regardless of whether the interposition of these third parities is desired by endpoints or not, it is not obvious to derive the reasoning about this situation from the end-to-end principle. To explain the rise of these third-party Intermediaries, we must either abandon the end-to-end principle or propose a new design principle that inter-operates the end-to-end principle. Clearly, the first option is infeasible since the end-to-end principle still plays a fundamental role of computer networks, which validates the need for another design principle.

1.1.5 The Asymmetric Capability of Endpoints

Created as a research network, the Internet was designed and used initially by technologists. As a result, the end-to-end principle offers perfect guide for networked systems since the technologists controlling the endpoints have full flexibility of designing and implementing applications without worrying about the underlining network. However, as the Internet witnesses its tremendous success, the majority of devices (endpoints) connected to the Internet are prebuilt boxes (*i.e.,* PCs or mobile devices) controlled by ordinary people. In such circumstance, the asymmetric capabilities of all endpoints reach an extreme where one party (*e.g.,* service providers, researchers) wishes to innovate the

network, for instance, to make it more reliable and secure, whereas the other party (*e.g.,* ordinary clients) simply may not have the ability of installing, configuring, maintaining or/and upgrading the software to implement these new designs accurately.

A good example lying in this dimension is that many academic research proposals designed around the end-to-end principle turn out to be difficult to deployed in the Internet. In particular, one major category of academic approaches of addressing the volumetric DDoS attacks is the capability-based scheme in which each client needs to carry a permission (capability) in order to send traffic to a destination [7]. These designs typically require clients to change their network stack to insert capabilities into a layer between the IP layer and transport layer such that the target server's network is secured with minimal support from intermediate verification points (no more complicated than typical packet forwarding). The difficulty of enforcement of an end-to-end design in the Internet also speaks for the need for another design principle in computer networks.

1.2 The Arguments of the New Principle

The above five dimensions are intended to illustrate the richness of new design requirements of computer networks. The existence of these requirements does not mean that end-to-end principle is no longer valid and should be abandoned. Instead, it shows that designing computer networks is becoming much more complex than it was when the end-to-end principle was first articulated several decades ago. As a result, to better meet these emerging requirements, we need a new principle for the design space that has not been explored by the end-to-end principle, such that a combination of the two principles provides more complete, effective and accurate guides for innovating the modern computer networks and their applications.

It is straightforward that the design space not explored by the end-to-end principle is how an in-network function should be implemented if it is a must. However, the self-evident design space in fact implies that it is non-trivial to propose valid new design principle since the network design has already been guided by, essentially, an opposite principle, *i.e.,* the end-to-end principle. It is highly possible that an improper new principle could offset or even neutral-

ize the success achieved by the end-to-end principle. Toward proposing valid design, we articulate the two primary arguments for the new principle.

1.2.1 Minimal Intermediary Deployment

The new design requirements lying in the first three dimensions described in Section 1.1 all lead to a call for adding new mechanism or function, besides the basic packet forwarding, inside the network core. Adding an in-network function is the most explicit challenge that the new principle imposes on the end-to-end principle. To preserve the power of the end-to-end principle as much as possible, the new principle argues for *minimal intermediary deployment* when implementing an in-network function that cannot be completely and accurately implemented only at the endpoints of the networked system.

The minimal intermediary deployment argument essentially speaks that the implementation of an in-network function should impose as little modification on the networked system as possible. There are many forms of modification to a networked system, including requiring upgrading the network core (*e.g.*, adding cryptographic support on Internet routers), requiring software upgrade from other endpoints, scarifying network performance (*e.g.,* reduced throughput or added latency) and weakening network properties (*e.g.*, reduced entropy of anonymous systems). Of course, in different networked systems, the willingness for accepting each form of modification can vary dramatically. For instance, weakening the privacy guarantee in Tor is much more disruptive than in the Internet, whereas upgrading Internet clients is far more difficult (as explained in Section 1.1.5) than that in Tor and datacenter networks. Regardless of how each form of modification is weighted, the minimal intermediary deployment argument states an bias toward implementing an in-network function while imposing as small change as possible on the networked system.

The followings are concrete examples of how the minimal intermediary deployment argument drives the system designs in this thesis. Consider MiddlePolice that is designed for addressing volumetric DDoS attacks in the Internet as the first example. Since the Internet is composed by over 60,000 Autonomous Systems (ASes) with varying levels of technological sophistication and cooperativeness, it would impose significant disruption if the implementation of an in-network function requires upgrades from a large majority of ASes. Thus, driven

by the minimal intermediary deployment argument, MiddlePolice's in-network function requires deployment only at the target server and possibly at related parties on commercial terms (*e.g.,* the cloud), making MiddlePolice to be readily deployable in the existing infrastructure of the Internet.

The Tor network is designed to protect the online privacy of Tor users. Under the principle of minimal intermediary deployment, any in-network function must be implemented without weakening the privacy guarantee offered by Tor. Towards this end, TorPolice's design to relies on anonymous identities and blind signatures to completely preserve Tor user privacy. Finally, in QShare and OpReduce designed for optimizing tenant networks to support their performance demanding applications, the minimal intermediary deployment argument drives their designs to leverage only existing features on the commodity switches so that the datacenter networks do not experience any performance degradation for implementing in-network functions.

1.2.2 Endpoint-Driven In-Network Function

The end-to-end principle is not offered as an absolute. Therefore, a number of in-network functions have been deployed inside the core network. One of the most widely adopted in-network functionalities is the security firewalls or middleboxes that are inserted into inside the network to protect their downstream networks. These intermediate servers inspect and analyze traversing traffic and enforce corresponding forwarding decisions after determining the intent or nature of the traffic. One of the most well-known parties that offer such security in-network functionality is the commercial DDoS-prevention service providers such as Arbor Networks, Akamai and Cloudflare. These service providers ask their customers to redirect customer traffic to their massively over-provisioned intermediate infrastructures, and then they apply a variety of techniques to scrub traffic, separating malicious from benign, and then re-injects only the benign traffic back to their customers. Another well-known in-network function is the various load balancing techniques (*e.g.,* ECMP) in datacenter networks to distribute traffic across multiple available paths to fully utilize the bisection datacenter bandwidth. In the Tor network, the immediate relays on Tor circuits even implement queue-based congestion controls so that each Tor circuit is given a fairshare of the relay's bandwidth.

The commonality of these existing in-network functionalities is that no endpoint intelligence and information are ever considered for the design and implementation of these functions. For instance, the DDoS-prevention service providers rely on empirical and proprietary filtering algorithms to discard traffic that they believe is malicious. As a result, the target server cannot prioritize desired traffic kinds or choose preferred scheduling policies during DDoS mitigation. The in-network load balancing functions in datacenter networks distributed flows, for instance, based on static hashing of the flow identifier (typically the five-tuple), without considering which flows are sent by which VMs (tenants). Although such in-network functions in general improve networking performance by reducing congestion, they are still incapable of optimizing tenant networks for their performance demanding applications. Finally, the congestion control currently implemented on Tor relays is to achieve local fairshare among circuits, which neither prevents attackers from gaining more resources by building more circuits nor makes any contributions to separate abusive traffic from legitimate one.

It is clear that existing in-network functions are far from being effective in meeting these emerging requirements. Fundamentally, any network core driven in-network functions would reach finally their limitations since the network only has limited knowledge about the endpoint applications. Consider, for example, the large DDoS-prevention vendors that monitor and measure huge amount of Internet traffic. Even after decades of operation, the best offering from these vendors is still empirical filtering, which is certainly insufficient against sophisticated and unprecedented attacks: based on our interviews with multiple customers (*e.g.,* hosting companies), they are attacked every single day even after purchasing services from these vendors.

To address above fundamental limitation, we argue that an in-network function should be designed in an open framework such that network endpoints can configure and customize the in-network function to achieve self-desired network forwarding. Essentially, the argument of endpoint-driven in-network function advocates for augmenting the implementation of a new in-network function with endpoint-specified policies. For instance, rather than using any network-defined filtering rules, MiddlePolice's in-network function mitigates DDoS attacks by forwarding traffic based on destination-driven policies, including accepting only privileged traffic sent with permissions to prevent network overwhelming by unwanted traffic, determining how much privileged

traffic is allowed from each source, from each AS or for solving one challenge to achieve a wide variety of fairness metrics, and designing specific policies for mission-critical traffic. Such destination-driven in-network defense not only enables the target server to minimize its possible disruption during DDoS mitigation, but also makes the defense itself more effective since the network only needs to deliver all victim-desired traffic without needing to engaging in an arms race with attackers to accurately identity attack traffic.

Similarly, in TorPolice, rather than relying on Tor relays to decide malicious or abusive traffic, which may impose additional privacy threats, TorPolice's in-network access control mechanism enables a website to determine, anonymously, the number of service requests allowed by any Tor client. As a result, the website can effectively throttle Tor-emitted abuse and meanwhile serve legitimate Tor clients properly. Finally, in both QShare and OpReduce, rather than distributing network traffic using any network-implemented load balancer, both systems design an in-network tenant routing control mechanism to accurately control the forwarding path of tenants. Augmented with a search and optimization decoupled framework, such in-network routing control is open to optimize tenant network for arbitrary performance metrics.

There are no fundamental limitations about what types of endpoint-driven policies are allowed and what types of policies are prohibited. In general, there is a tradeoff between the complexity or/and benefit of endpoint-driven policies and the cost of enforcing them by in-network functions. For instance, one extreme case is that the implementation of a specific policy is so heavyweight that it even violates the argument of minimal intermediary deployment. Therefore, all endpoint-driven policies proposed in this thesis strike a balance among being general, being effective to address a specific challenge and being cost efficient. For instance, in MiddlePolice, one of the primary policies designed and implemented is per-sender fairshare. This policy has a high level of generality since it may be desired by a fairly large number of target servers and it can be easily customized, for instance, by classifying some senders as privileged ones. Second, this policy is effective against DDoS attacks since it offers guaranteed bandwidth share for legitimate clients, which is in fact the optimal policy against strategic attackers that can behave the exactly same as legitimate users (therefore the target servers cannot differentiate them). Finally, the implementation of the per-sender fairshare policy introduces reasonable cost, *e.g.,* the network throughput declines less than 9% even against massive scale

DDoS attacks involving hundreds of thousands bots.

Similarly, TorPolice considers a policy that allows a website to control the service request rate by any Tor client using self-desired parameters. Clearly, the policy is general as the parameters are defined by the website. Meanwhile it is effective since a website can properly set its parameters so as to bound the service request rate by any strategic adversary and therefore limit Tor-emitted abuse. Finally, the implementation cost is reasonable, comparing with the existing cost of setting up Tor circuits. Both QShare and OpReduce have the same balance: (i) they are designed as a general optimization framework that can optimize a variety of performance metrics for tenant applications, (ii) they are effective due to the search and optimization decoupled design and (iii) they introduce negligible overhead since their implementations leverage existing built-in features of commodity hardware.

1.3 Thesis Statement

Based on the above discussion, in this thesis, we claim the following statement is true.

If implementing an in-network function is a must for meeting an emerging requirement in networked systems and their running applications, the in-network function should be implemented with minimal intermediary deployment and its design should be augmented with endpoint-driven policies.

1.4 Thesis Organization

In this thesis, based on above principle, we design four systems in three different networked systems to address four emerging challenges. In Section 1.2, we have briefly explained how the design of each system is driven by the principle. In each of the following four chapters, we will elaborate on each of these systems, Specifically,

- In Chapter 2, we propose MiddlePolice, the first readily deployable DDoS defense mechanism that can enforce a wide variety of victim-selectable policies during DDoS mitigation

- In Chapter 3, we propose TorPolice, the first privacy-preserving access control framework for Tor that enables service providers to effectively throttle Tor-emitted abuses and meanwhile serving legitimate Tor clients.
- In Chapter 4, we propose QShare, a complete in-network solution to achieve work-conserving bandwidth guarantees in multi-tenant datacenters, which not only offers predictable performance for tenants, but also achieves efficient network resource utilization compared with static reservation.
- In Chapter 5, we propose OpReduce, a virtual tenant network management framework for optimizing tenant networks in multi-tenant datacenters.

Finally, Chapter 6 concludes this thesis.

CHAPTER 2

TOWARD ENFORCING DESTINATION-DEFINED POLICIES IN THE MIDDLE OF THE INTERNET

2.1 Introduction

Attacks against availability, such as distributed denial of service attacks (DDoS), continue to plague the Internet. The most common of these attacks, representing roughly 65% of all DDoS attacks in 2015 [8], are volumetric attacks. In these attacks, adversaries seek to deny service by exhausting a victim's network resources and causing congestion. Such attacks are difficult for a victim network to mitigate as the largest of these attacks can exceed the available upstream bandwidth by orders of magnitude. For example, Internet service providers (ISP) reported attacks in excess of 500 Gbps in 2015 [8].

One common solution to this problem is the use of DDoS-protection-as-a-service providers, such as CloudFlare. These providers massively overprovision datacenters for peak attack traffic loads and then share this capacity across many customers as needed. When under attack, victims use DNS or BGP to redirect traffic to the provider rather than their own networks. The DDoS-protection-as-a-service provider applies a variety of techniques to scrub this traffic, separating malicious from benign, and then re-injects only the benign traffic back into the network to be carried to the victim. Such methods are appealing, as they require no modification to the existing network infrastructure and can scale to handle very large attacks. However, these cloud-based systems use proprietary attack detection algorithms and filtering which limit the ability of customers to prioritize traffic kinds or choose preferred scheduling policies. Further, existing cloud-based systems assume that all traffic to the victim will be routed first to their infrastructure, an assumption that can be violated by a clever attacker [9, 10].

A second approach to solving volumetric DDoS attacks is network capability-based solutions [7, 11–13]. Such systems require a source to receive explicit

permission before being allowed to contact the destination. Such capabilities are enforced by the network infrastructure itself (*i.e.,* routers) and capabilities range from giving the victim the ability to block traffic from arbitrary sources to giving the victim control over the bandwidth allowed for each flow. A major advantage, then, of these capability-based systems is the ability of the victim to control precisely what and how much traffic it wants to receive. However, these capability-based systems are not without challenges, and most face significant deployment hurdles. For instance, approaches such as SIFF [11], TVA [12] and NetFence [13] require secret key management and router upgrades across different Autonomous Systems (ASes). Yet other approaches require clients to modify their network stack to insert customized packet headers, creating additional deployment hurdles.

In this proposal, we present MiddlePolice, which seeks to combine the deployability of cloud-based solutions with the destination-based control of capability-based systems. MiddlePolice is built on a set of traffic policing units (referred as mboxes) which rely on a feedback loop of self-generated capabilities to guide scheduling and filtering. MiddlePolice also includes a mechanism to filter nearly all traffic that tries to bypass the mboxes, using only the ACL configuration already present on commodity routers. We implement MiddlePolice as a Linux kernel module, and evaluate it extensively over the Internet using cloud infrastructures, on our private testbed, and via simulations. Our results show that MiddlePolice can handle large-scale DDoS attacks, and effectively enforce the destination-chosen policies.

In MiddlePolice, underpinning is a novel and robust network capability design. To begin with, the capability design itself incorporates a Message Authentication Code (MAC) so that the adversaries without valid keys cannot sabotage the system. Functionally, the capabilities allow the mboxes to learn downstream bandwidth availability without additional deployment, which is the key to remove deployment from irrelevant network entities. Atop capabilities, various bandwidth sharing algorithms are designed and implemented at mboxes to faithfully enforce the victim-defined policies.

This chapter elaborates on the design of MiddlePolice, including the capability generation and traffic policing algorithms. A prototype built atop the existing Internet infrastructure demonstrates MiddlePolice's immediate deployability. Finally, extensive evaluations on both physical testbed and the ns-3 simulator are performed to prove MiddlePolice's effectiveness to stop large-scale DDoS

attacks.

2.2 Problem Formulation

2.2.1 MiddlePolice's Properties

Readily Deployable and Scalable. MiddlePolice is designed to be readily deployable in the Internet and sufficiently scalable to handle large-scale attacks. *To be readily deployable, a system should only require deployment at the destination, and possibly at related parties on commercial terms.* The end-to-end principle of the Internet, combined with large numbers of endpoints, is what gives rise to its tremendous utility. Because of the diversity of administrative domains, including endpoints, edge-ASes, and small transit ASes, ASes have varying levels of technological sophistication and cooperativeness. However, some ASes can be expected to help with deployment; many ISPs already provide some sort of DDoS-protection services [14], so we can expect that such providers would be willing to deploy a protocol under commercially reasonable terms. We contrast this with prior capability-based work, which requires deployment at a large number of unrelated ASes in the Internet and client network stack modification, that violates the deployability model.

The goal of being deployable and scalable is the major reason that MiddlePolice is designed to be built into existing cloud-based DDoS defense systems.

Destination-Driven Policies. MiddlePolice is designed to provide the destination with fine-grained control over the utilization of their network resources. Throughout this chapter, we use "destination" and "victim" interchangeably. Existing cloud-based systems have not provided such functionality. Many previously proposed capability-based systems are likewise designed to work with a single scheduling policy. For instance, NetFence [13] enforces per-sender fairness, CRAFT [15] enforces per-flow fairness, Portcullis [16] and Mirage [17] enforce per-compute fairness, SIBRA [18] enforces per-steady-bandwidth fairness, and SpeakUp [19] enforces per-outbound-bandwidth fairness. If any of these mechanisms is ever deployed, a single policy will be enforced, forcing the victim to accept the choice made by the defense approach. However, no single fairness regime can satisfy all potential victims' requirements. Ideally, Middle-

Police should be able to support victim-chosen policies. In addition to these fairness metrics, MiddlePolice can implement ideas such as ARROW's [20] special pass for critical traffic, and prioritized services for premium clients.

Fixing the Bypass Vulnerability. Existing cloud-based systems rely on DNS or BGP to redirect the destination's traffic to their infrastructures. However, this model opens up the attack of infrastructure bypass. For example, a majority of cloud-protected web servers are subject to IP address exposure [9, 10]. Larger victims that join in the Shared Whois Project (SWIP) [21] may be unable to keep their IP addresses secret from a determined adversary. In such cases, the adversary can bypass the cloud infrastructures by routing traffic directly to the victims. MiddlePolice includes a readily deployable mechanism to address this vulnerability.

MiddlePolice is designed to augment the existing cloud-based DDoS prevention systems with destination-selectable policies. The literature is replete with capability-based systems that provide a single fairness guarantee with extensive client modification and deployment at non-affiliated ASes. The novelty and challenge of MiddlePolice is therefore architecting a system to move deployment to the cloud while enforcing a wide variety of destination-selectable fairness metrics. Built atop a novel capability feedback mechanism, MiddlePolice meets the challenge, thereby protecting against DDoS more flexibly and deployably.

2.2.2 Adversary Model and Assumptions

Adversary Model. We consider a strong adversary owning large botnets that can launch strategic attacks and amplify its attack [22]. We assume the adversary is not on-path between any mbox and the victim, since otherwise it could drop all packets. Selecting routes without on-path adversaries is an orthogonal problem and is the subject of active research in next-generation Internet protocols (*e.g.,* SCION [23]).

Well-Connected mboxes. MiddlePolice is built on a distributed and replicable set of mboxes that are well-connected to the Internet backbone. We assume the Internet backbone has sufficient capacity and path redundancy to absorb large volumes of traffic, and DDoS attacks against the set of all mboxes can never be successful. This assumption is a standard assumption for cloud-based systems.

Victim Cooperation. MiddlePolice's defense requires the victim's cooperation. If the victim can hide its IP addresses from attackers, it simply needs to remove a MiddlePolice-generated capability carried in each packet and return it back to the mboxes. The victim needs not to modify its layer-seven applications as the capability feedback mechanism is transparent to applications. If attackers can directly send or point traffic (*e.g.,* reflection) to the victim, the victim needs to block the bypassing traffic. MiddlePolice includes a packet filtering mechanism that is immediately deployable on commodity Internet routers.

Cross-Traffic Management. We assume that bottlenecks on the path from an mbox to the victim that is shared with other destinations are properly managed, such that cross-traffic targeted at another destination cannot cause unbounded losses of the victim's traffic. Generally, per-destination-AS traffic shaping (*e.g.,* weighted fair share) on these links will meet this requirement.

2.3 System Overview

MiddlePolice's high-level architecture is illustrated in Figure 2.1. A MiddlePolice-protected victim uses a DNS entry to redirect its traffic to the mboxes. Each mbox polices traversing traffic, enforcing the sharing policy chosen by the victim. This policing relies on a feedback loop of MiddlePolice-generated capabilities to eliminate the need for deployment at downstream routers. When the victim keeps its IP addresses secret (CloudFlare [24], for example, assumes the same model), a single deploying mbox can secure the entire path from the mbox to the victim. Deployments of distributed mboxes close to the Internet core, for instance by outsourcing them to the cloud, can therefore police traffic before it reaches downstream bottlenecks. Further, since tasks performed by each mbox is trivially parallelizable, MiddlePolice can scale through the replication of mboxes at diverse geographical and network locations.

Larger victims that SWIP [21] their IP addresses may be unable to keep their IP addresses secret from a determined adversary. As a result, attackers can bypass the mboxes and send attack traffic directly to the victim. MiddlePolice designs a novel packet filtering mechanism relying on the ACL on deployed routers or switches to eliminate the traffic that does not traverse any mbox. As long as each bottleneck link is protected by an upstream filter, direct attack can be prevented.

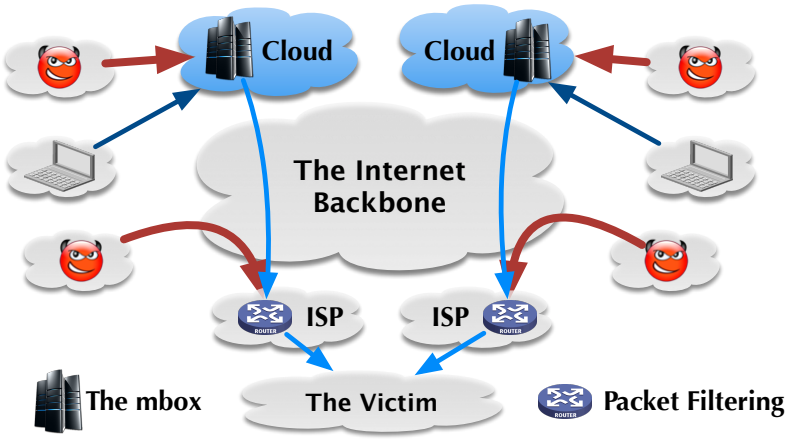


Figure 2.1: The architecture of MiddlePolice to protect a dual-homed victim.

2.4 Detailed Design of mboxes

MiddlePolice's traffic policing algorithm (i) probes the available downstream bandwidth from each mbox to the victim and (ii) allocates the bandwidth to senders according to the fairness regimes chosen by the victim.

Bandwidth Probe. The fundamental challenge of estimating downstream bandwidth is that MiddlePolice requires no deployment at downstream links. Such a challenge is two-sided: an overestimate will cause downstream flooding, rendering traffic policing useless, while an underestimate will waste downstream capacity, reducing performance.

To solve the overestimation problem, MiddlePolice relies on a *capability feedback* mechanism to make senders self-report how many packets they have successfully delivered to the victim. Specifically, upon a packet arrival, the mbox stamps an unforgeable capability in the packet. When the packet is delivered to the victim, MiddlePolice's capability handling module (CHM) deployed on the victim is responsible to return the carried capability back to the mbox. If the capability is not fed back to the mbox after a sufficiently long time interval (compared with the RTT from the mbox to the victim), the mbox will consider that the packet has been lost. Thus, the feedbacks enable the mbox to infer a packet loss rate (hereafter referred as LLR) for each sender. Then the downstream capacity is estimated as the difference between the number of arrived packets from all senders and lost packets on the downstream path. As the estimation is based on the traffic volume *delivered* to the victim, it solves overestimation problem.

However, the above technique is not sufficient to overcome the underesti-

mation problem. Specifically, because sender demand may be less than downstream capacity, simply using the volume of delivered traffic may cause underestimation. To solve the problem, the mbox categorizes packets from each sender as *privileged packets* and *best-effort* packets. Specifically, the mbox maintains a rate window $\mathcal{W}_R$ for each sender to determine the amount of privileged packets allowed for the sender in each period (hereafter referred *detection period*). $\mathcal{W}_R$ is computed based on the above downstream capacity estimation as well as the sharing policies chosen by the victim. Packets sent beyond $\mathcal{W}_R$ are classified as best-effort packets. The mbox forwards all privileged packets to the victim, whereas the forwarding decisions for best-effort packets are subject to a short-term packet loss rate (hereafter referred as SLR). The SLR reflects downstream packet loss rates (congestion) at a RTT granularity. That is, if the downstream is not congested upon an arrival of a best-effort packet, the mbox will forward the packet. Thus, even when the downstream capacity is underestimated (the $\mathcal{W}_R$ is underestimated), the mbox can still further deliver packets as long as the downstream path is not congested.

Fairness Regimes. Each mbox allocates its probed bandwidth to senders based on the sharing policies chosen by the victim. For policies enforcing global fairness among all senders, the mboxes sharing the same bottleneck may need to share their local observations. We address the problem of co-bottleneck detection in Section 2.8.3.

2.4.1 Information Table

The basis of MiddlePolice's traffic policing is an information table (*iTable*) maintained by each mbox. Each row of the *iTable* corresponds to a single sender. The contents of the *iTable* depend on the victim-selected sharing policy; this section describes *iTable* elements needed for per-sender fairness, and Section 2.4.3.5 extends the *iTable* to other fairness regimes. In Section 2.6, we describe a mechanism to filter source spoofing at the mbox, so this section ignores source spoofing.

Table 2.1: Fields of an *iTable* entry and their sizes (bits).

f	$\mathcal{T}_A$	$\mathcal{P}_{id}$	$\mathcal{N}_R$	$\mathcal{N}_D$	$\mathcal{W}_R$	$\mathcal{W}_V$	$\mathcal{L}_R$
64	32	16	32	32	32	128	64

Each sender s_i has one row in the *iTable*, identified by a unique identifier f. The fields and their corresponding sizes in each entry are illustrated in Table 2.1. Other than f, each field is updated each detection period. The timestamp $\mathcal{T}_A$ records the current detection period. The capability ID $\mathcal{P}_{id}$ is the maximum number of distinct capabilities generated for s_i. $\mathcal{N}_R$ stores the number of packets received from s_i. $\mathcal{N}_D$ indicates the number best-effort packets dropped by the mbox. $\mathcal{W}_R$ determines the maximum number of privileged packets allowed for s_i. The verification window $\mathcal{W}_V$ is designed to compute s_i's packet loss rate, whereas $\mathcal{L}_R$ stores the LLR for s_i.

2.4.2 Capability Computation

For s_i, the mbox generates two types of capabilities: distinct capabilities and common capabilities. The CHM can use either capability for authentication, whereas only distinct capabilities are used to infer downstream packet losses.

A distinct capability for s_i is computed as follows:

$$\begin{aligned} \mathcal{C} = IP_{MP} \,||\, ts \,||\, \mathcal{P}_{id} \,||\, f \,||\, \mathcal{T}_A \,||\, \\ MAC_{K_s}(IP_{MP} \,||\, ts \,||\, \mathcal{P}_{id} \,||\, f \,||\, \mathcal{T}_A), \end{aligned} \tag{2.1}$$

where IP_{MP} is the IP address of the mbox issuing $\mathcal{C}$ and ts is the current timestamp to mitigate replay attack. The combination of $\mathcal{P}_{id}||f||\mathcal{T}_A$ ensures the uniqueness of $\mathcal{C}$. The MAC is computed based on a secret key K_s shared by all mboxes. The MAC is 128 bits, so the entire $\mathcal{C}$ consumes ~300 bits.

A common capability, defined in Equation (2.2), carries a MAC for packet authentication.

$$\mathcal{C}_c = IP_{MP} \,||\, ts \,||\, MAC_{K_s}(IP_{MP} \,||\, ts). \tag{2.2}$$

The design of capability incorporates a MAC to ensure that attackers without secure keys cannot generate valid capabilities, preventing capability abuse.

Table 2.2: System parameters.

Para.	Definition	Value
$\mathcal{D}_p$	The length of the detection period	$4s$
Th_{cap}	Distinct capability ID upper bound	128
Th_{rrt}	Maximum waiting time for cap. feedbacks	1s
Th_{slr}^{drop}	SLR thres. for dropping best-effort pkts	0.05
β	The weight for historical loss rates	0.8
Th_{lpass}	The threshold for calculating LLR	5
S_{slr}	The length limit of *cTable*	100

2.4.3 Traffic Policing Logic

2.4.3.1 Populating the *iTable*

We first describe how to populate the *iTable*. At time ts, the mbox receives the first packet from s_i. It creates an entry for s_i, with f computed based on s_i's source address, and initializes the remaining fields to zero. It then updates $\mathcal{T}_A$ to ts, increases both $\mathcal{N}_R$ and $\mathcal{P}_{id}$ by one to reflect the packet arrival and computes a capability specific to the updated $\mathcal{P}_{id}$ and $\mathcal{T}_A$.

Upon receiving a packet from s_i with arrival time $t_a - \mathcal{T}_A > \mathcal{D}_p$ (t_a is the arrival time and $\mathcal{D}_p$ is the length of the detection period), the mbox starts a new detection period for s_i by setting $\mathcal{T}_A = t_a$. The mbox also updates the remaining fields based on the traffic policing algorithm (as described in Section 2.4.3.4). As the algorithm depends on s_i's LLR and the mbox's SLR, the computation of which is described in the following sections.

2.4.3.2 Inferring LLR

Capability Generation. For each packet from s_i, the mbox generates a distinct capability for the packet if (i) its arrival time $t_a - \mathcal{T}_A < \mathcal{D}_p - Th_{rtt}$, and (ii) the capability ID $\mathcal{P}_{id} < Th_{cap}$. The first constraint ensures that the mbox allows at least Th_{rtt} for each capability to be returned from the CHM. By setting Th_{rtt} well above the RTT from the mbox to the victim, any missing capabilities at the end of the current period correspond to lost packets. Table 2.2 lists the system parameters including Th_{rtt} and some suggested values.

The second constraint Th_{cap} is to bound the number of distinct capabilities issued for s_i in one detection period, so as to bound the memory requirements.

We set $Th_{cap} = 128$, large enough to reduce the sampling error in computing the LLR.

Packets violating any of the two constraints, if any, will carry a common capability (Equation (2.2)). The common capability will not be returned by the CHM to learn s_i's LLR. However, it can be used for packet authentication.

Capability Feedback Verification. Assume the mbox generates K_{th} distinct capabilities for s_i with capability ID ranging from $[1, K_{th}]$. Each time a capability is returned, the mbox checks the capability ID to determine which packet (carrying the received capability) has been received by the CHM. $\mathcal{W}_V$ represents a window with Th_{cap} bits. Initially all the bits are set to zero. When a capability with capability ID i is received, the mbox sets the ith bit in $\mathcal{W}_V$ to one. At the end of the current detection period, the zero bits in the first K_{th} bits of $\mathcal{W}_V$ indicate the losses of the corresponding packets. To reduce memory consumption, the mbox only processes capabilities issued in the current period.

LLR Computation. LLR in kth detection period is computed at the end of the period, *i.e.,* the time when the mbox decides to start a new detection period for s_i. s_i's lost packets may contain downstream losses and dropped best-effort packets by the mbox. The number of packets that s_i sent to downstream links is $\mathcal{N}_R - \mathcal{N}_D$, and the downstream packet loss rate is $\frac{V_0}{\mathcal{P}_{id}}$, where V_0 is the number of zero bits in the first $\mathcal{P}_{id}$ bits of $\mathcal{W}_V$. Thus, the estimated number of downstream packet losses is $\mathcal{N}_{loss}^{dstream} = (\mathcal{N}_R - \mathcal{N}_D)\frac{V_0}{\mathcal{P}_{id}}$. Then we have $LLR = (\mathcal{N}_{loss}^{dstream} + \mathcal{N}_D)/\mathcal{N}_R$.

Our strawman design is subject to statistical bias, and may have negative effects on TCP timeouts. In particular, assume one legitimate TCP source recovers from a timeout and sends one packet to probe the network condition. If the packet is dropped again, the source will enter longer timeouts. However, the source would have a (biased) 100% loss rate based on the strawman design. Adding a low-pass filter can fix this problem. Specifically, if s_i's $\mathcal{N}_R$ in the current period is less than a small threshold Th_{lpass}, the mbox sets its LLR in the current period as zero. In Section 2.4.3.4, we explain why the low pass filter is robust. Formally, we have

$$LLR = \begin{cases} 0, & \text{if } \mathcal{N}_R < Th_{lpass} \\ \frac{\mathcal{N}_{loss}^{dstream} + \mathcal{N}_D}{\mathcal{N}_R}, & otherwise. \end{cases} \tag{2.3}$$

2.4.3.3 Inferring SLR

SLR is designed to reflect downstream congestion on a per-RTT basis, and is computed across all flows from the mbox to the victim. Like LLR, SLR is learned through capabilities returned by the CHM. Specifically, the mbox maintains a hash table (*cTable*) to record capabilities used to learn its SLR. The hash key is the capability itself and the value is a single bit value (initialized as zero) to indicate whether the corresponding key (capability) has been returned.

As described in Section 2.4.3.2, when a packet arrives (from any source), the mbox stamps a capability on the packet. The capability will be added into *cTable* if it is not a common capability and *cTable*'s length has not reach a predefined threshold $\mathcal{S}_{slr}$. The mbox maintains a timestamp $\mathcal{T}_{slr}$ when the last capability is added into *cTable*. Then, it uses the entire batch of capabilities in *cTable* to learn the SLR. We set $\mathcal{S}_{slr} = 100$ to allow fast capability loading to the *cTable*, while minimizing sampling error from $\mathcal{S}_{slr}$ being too small.

As in LLR, the mbox allows at most Th_{rtt} from $\mathcal{T}_{slr}$ to receive all feedbacks in *cTable*. Note some capabilities may be returned before Th_{rtt} (*i.e.,* before *cTable* is full). Once a capability in *cTable* is returned by the CHM, the mbox sets its hash value to one. Upon receiving a new packet with arrival time $t_a > \mathcal{T}_{slr} + Th_{rtt}$, the mbox computes $SLR = \frac{Z_0}{\mathcal{S}_{slr}}$, where Z_0 is the number of *cTable* entries whose value is zero. The mbox then clears the current *cTable* entries to start a new monitoring cycle for SLR.

2.4.3.4 Traffic Policing Algorithm

We formalize the traffic policing logic in Algorithm 1. Upon receiving a packet P, the mbox retrieves the entry $\mathcal{F}$ in *iTable* matching P (Line 3). If no entry matches, the mbox initializes an entry.

P is categorized as a privileged or best-effort packet based on $\mathcal{F}$'s $\mathcal{W}_R$ (line 5). If P is privileged, the mbox performs necessary capability handling (line 6) before appending P to the privileged queue. The mbox maintains two FIFO queues to serve all *accepted* packets: the privileged queue serving privileged packets and the best-effort queue serving best-effort packets. The privileged queue has strictly higher priority than the best-effort queue. CapabilityHandling (line 11) executes the capability generation and *cTable* updates (line 32), as detailed in Section 2.4.3.2 and Section 2.4.3.3.

Algorithm 1: Traffic policing algorithm.

Main Procedure:
begin
  $\mathcal{F} \leftarrow$ iTableEntryRetrieval(P);
  $\mathcal{F}.\mathcal{N}_R \leftarrow \mathcal{F}.\mathcal{N}_R + 1$;
  if $\mathcal{F}.\mathcal{N}_R < \mathcal{F}.\mathcal{W}_R$ **then**
    /* Priviledged packets */
    CapabilityHandling($P, \mathcal{F}$);
    Append P to the privileged queue;
  else
    /* Best-effort packets */
    BestEffortHandling($P, \mathcal{F}$);
  /* Starting a new detection period if necessary */
  if $ts - \mathcal{F}.\mathcal{T}_A > \mathcal{D}_p$ **then** iTableHandling($\mathcal{F}$);

Function: CapabilityHandling($P, \mathcal{F}$):
begin
  /* Two constraints for capability generation */
  if $\mathcal{F}.\mathcal{P}_{id} < Th_{cap}$ ***and*** $ts - \mathcal{F}.\mathcal{T}_A < \mathcal{D}_p - Th_{rtt}$ **then**
    $\mathcal{F}.\mathcal{P}_{id} \leftarrow \mathcal{F}.\mathcal{P}_{id} + 1$;
    Generate capability $\mathcal{C}$ based on Equation (2.1);
    cTableHandling($\mathcal{C}$);
  else
    /* Common capability for packet authentication */
    Generate capability $\mathcal{C}_c$ based on Equation (2.2);

Function: BestEffortHandling($P, \mathcal{F}$):
begin
  if $SLR < Th_{slr}^{drop}$ ***and*** $\mathcal{F}.\mathcal{L}_R < Th_{slr}^{drop}$ **then**
    CapabilityHandling($P, \mathcal{F}$);
    Append P to the best-effort queue;
  else
    Drop P; $\mathcal{F}.\mathcal{N}_D \leftarrow \mathcal{F}.\mathcal{N}_D + 1$;

Function: iTableHandling($\mathcal{F}$):
begin
  Compute $recentLoss$ based on Equation (2.3);
  /* Consider the historical loss rate */
  $\mathcal{F}.\mathcal{L}_R \leftarrow (1-\beta) \cdot recentLoss + \beta \cdot \mathcal{F}.\mathcal{L}_R$;
  BandwidthSharingPolicy($\mathcal{F}$);
  Reset $\mathcal{W}_V$, $\mathcal{P}_{id}$, $\mathcal{N}_R$ and $\mathcal{N}_D$ as zero;

Function: cTableHandling($\mathcal{C}$):
begin
  /* One batch of *cTable*'s is not ready */
  if *cTable.length* $< \mathcal{S}_{slr}$ **then**
    Add $\mathcal{C}$ into *cTable*;
    if *cTable.length* $== \mathcal{S}_{slr}$ **then** $\mathcal{T}_{slr} \leftarrow ts$;

If P is a best-effort packet, its forwarding decision is subject to the SLR and $\mathcal{F}'s$ LLR (line 19). If the SLR exceeds Th_{slr}^{drop}, indicating downstream congestion, the mbox discards P. Further, if $\mathcal{F}$'s LLR is already above Th_{slr}^{drop}, the mbox will not deliver best-effort traffic for $\mathcal{F}$ as well since $\mathcal{F}$ already experiences severe losses. Th_{slr}^{drop} is set to be few times larger than a TCP flow's loss rate in normal network condition [25] to absorb burst losses. If the mbox decides to accept P, it performs capability handling as necessary (line 22).

Finally, if P's arrival triggers a new detection period for $\mathcal{F}$ (line 10), the mbox performs corresponding updates for $\mathcal{F}$ (line 26). To determine $\mathcal{F}$'s LLR, the mbox incorporates both the recent LLR ($recentLoss$) obtained in the current period and $\mathcal{F}$'s historical loss rate $\mathcal{L}_R$. Such a design prevents attackers from hiding their previous packet losses via on-off attacks. Similarly, adding a low pass filter in Equation (2.3) is not subject to abuse. $\mathcal{F}$'s $\mathcal{W}_R$ is updated based on the sharing policy proposed by the victim (line 30), as described below.

2.4.3.5 Bandwidth Sharing Policies

We list the following representative policies that may be chosen to implement in BandwidthSharingPolicy.

NaturalShare presents a simple policy: for each sender, the mbox sets its $\mathcal{W}_R$ for the next detection period as the number of delivered packets in the current period. The design rationale is that the mbox allows a rate that the sender can sustainably transmit without experiencing a large LLR.

PerSenderFairshare allows the victim to enforce per-sender fair share at bottlenecks. Each mbox fairly allocates its estimated total downstream bandwidth to all serving senders. By maintaining a global value $\mathcal{N}_{size}^{total}$ to store the total capacity estimate, and updating each sender's contribution to the $\mathcal{N}_{size}^{total}$ each time the sender starts a new period, the mbox can obtain the fair rate for each sender via a single entry access rather than traversing through the entire *iTable*.

To ensure the global fairness among all senders, the mboxes sharing the same bottleneck need to share their local observations. We design a co-bottleneck detection mechanism based on the SLR correlation of the mboxes: if two mboxes' SLRs are correlated, they share a bottleneck with high probability. In Section 2.8.3, we validate the effectiveness of the mechanism.

PerASFairshare is similar to PerSenderFairshare except that the mbox fairly allocates the bandwidth on a per-AS basis. This policy mimics SIBRA [18], pre-

venting bot-infested ASes from taking bandwidth away from legitimate ASes.

PerASPerSenderFairshare is a hierarchical fairness regime: the mbox first allocates its bandwidth on a per-AS basis, and then fairly assigns the bandwidth obtained by each AS among the AS's senders.

2.5 Packet Filtering

When the victim's IP addresses are kept secret, attackers cannot bypass MiddlePolice's upstream mboxes to route attack traffic directly to the victim. In this case, the downstream packet filtering mechanism is unnecessary since MiddlePolice can throttle attack traffic at the upstream mboxes. However, with tools like WHOIS [26], it may be difficult for a large victim owning a large chunk of IP addresses to keep all of them secret. In this case, the victim needs to deploy a packet filtering mechanism to filter out packets that have not traversed MiddlePolice's mboxes.

Filtering Primitives. Although the MAC-incorporated capability can prove that a packet indeed traverses an mbox, it requires upgrades from deployed commodity routers to perform MAC computation to filter bypassing packets. Thus, we invent a mechanism based on the existing ACL configurations on commodity routers. Specifically, each mbox encapsulates its traversing packets into UDP packets (similar techniques have been applied in VXLAN and [27]), and uses the UDP source and destination ports (a total of 32 bits) to carry an authenticator, a shared secret between the mbox and the filtering point. Although a 32-bit secret is not long enough to prevent attackers from generating the secret, a 400 Gbps attack (the largest attack viewed by Arbor Networks [8]) that uses random port numbers will be reduced to ~90 bps as the chance of a correct guess is one over 2^{32}. The shared secret can be negotiated periodically based on a cryptographically secure pseudo-random number generator. Note that although the UDP source address can also be adopted for filtering, attackers may spoof the mbox's source address.

Packet Filtering Points. The filtering mechanism should be deployed at, or upstream of, each bottleneck link. Deployed filtering points should have sufficient bandwidth so that the bypassing attack traffic cannot cause packet losses prior to the filtering. To ensure this, the victim can either purchase sufficient bandwidth from large ISPs and deploy the filtering mechanism at the inbound points

of its network. Or it can work with its ISP to deploy the filtering deeper in the ISP's network. Working with the victim's ISP does not violate the deployment model in Section 2.2 as MiddlePolice never requires deployment from unrelated ASes.

2.6 Source Validation

The design of MiddlePolice relies on the ability to punish sources that have excessive LLRs. In order for such punishment to be effective and correctly directed, we must limit an attacker's ability to create many new sources. MiddlePolice could build on previous work (*e.g.,* [16]) that ensures fairness or accountability in the setup process; however, such work often includes mechanisms that are difficult to deploy. As a result, we also design a source verifier for MiddlePolice to ensure that a sender is on-path to its claimed source IP address. The verifier is completely transparent to clients, and is enforced entirely at the mboxes.

Our key insight is that the HTTP Host header is in the first few packets of each connection. As a result, the mbox monitors a TCP connection and reads the Host header. If the Host header reflects a generic (not sender-specific) hostname (*e.g.,* victim.com), the mbox intercepts this flow, and redirects the connection (HTTP 302) to a Host name containing a token cryptographically generated from the sender's claimed source address, *e.g., T*.victim.com, where *T* is the token. If the sender is on-path, it will receive the redirection, and its further connection will use the sender-specific hostname in the Host header. When an mbox receives a request with a sender-specific Host, it verifies that the Host is proper for the claimed IP source address (if not, the mbox initiates a new redirection), and forwards the request (and any keys established by HTTPS) to the victim. TCP Fast Open can be adopted between the mbox and the victim to reduce the handover latency; because traffic from the mbox is authenticated, Denial-of-Service concerns of TCP Fast Open can be mitigated.

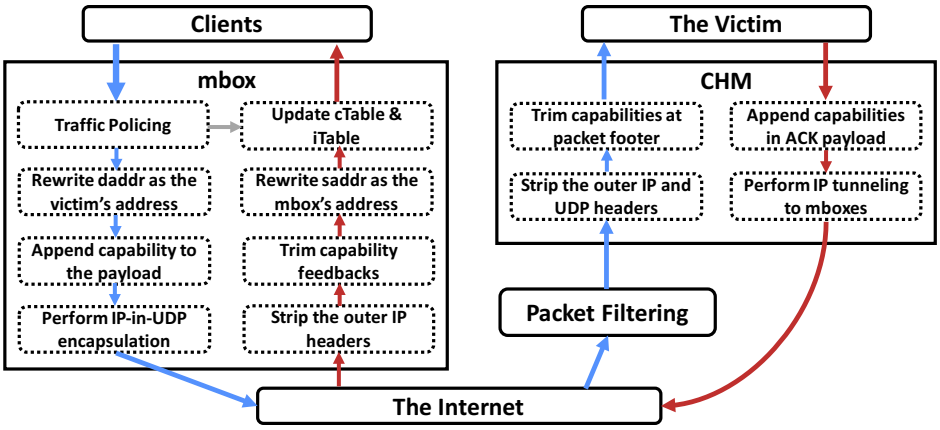


Figure 2.2: The software stack of the mbox and CHM.

2.7 Implementation

In this section, we discuss the implementation of MiddlePolice, starting with the and the capability handling module (CHM).

2.7.1 mboxes and CHM Implementation

The mboxes and the CHM at the victim are implemented based on the NetFilter Linux Kernel Module, which combined have ~1500 lines of C code (excluding the capability generation code). The software stack is illustrated in Figure 2.2. We elaborate each component following the packet handling process.

All inbound traffic from clients to an mbox is subject to the traffic policing whereas only accepted packets go through the capability-related processing. Packet dropping due to traffic policing triggers *iTable* updates. For each accepted packet, the mbox rewrites its destination address as the victim's address to point the packet to the victim. To carry capabilities, rather than defining a new packet header, the mbox appends the capabilities to the end of the original data payload, which avoids compatibility problems at intermediate routers and switches. The CHM is responsible to trim these capabilities to deliver the original payload to the victim's applications. If the packet filter is deployed, the mbox performs the IP-in-UDP encapsulation, and adopts the UDP source and destination ports to carry the secret. Note that all checksums need to be recomputed after packet manipulation to ensure correctness.

To avoid packet fragmentation due to the additional 68 bytes added by the mbox (20 bytes for outer IP header, eight bytes for the outer UDP header and

40 bytes reserved for a capability), the mbox needs to be a priori aware of the MTU M_d on its path to the victim. Then the mbox sets its MSS as no more than $M_d - 68 - 40$ (the MSS is 40 less than the MTU). Note that we do not directly set the MTU of the mbox's NIC to rely on the path MTU discovery to find the right packet size because ISPs may block ICMP packets. One our testbed, as $M_d = 1500$, we set the mbox's MSS as 1360.

Upon receiving packets from upstream mboxes, the CHM strips their outer IP/UDP headers and trims the capabilities. To return these capabilities, the CHM piggybacks capabilities to the payload of ACK packets. To ensure that a capability is returned to the mbox issuing the capability even if the Internet path is unsymmetrical, the CHM performs IP-in-IP encapsulation to tunnel the ACK packets to the right mbox. We allow one ACK packet to carry multiple capabilities since the victim may generate cumulative ACKs rather than per-packet ACKs. Further, the CHM tries to pack more capabilities in one ACK packet to reduce the capability feedback latency at the CHM. The number of capabilities carried in one ACK packet is stored in the TCP option (the 4-bit `res1` option). Thus, the CHM can append up to 15 capabilities in one ACK packet if the packet has enough space and the CHM has buffered enough capabilities.

Upon receiving an ACK packet from the CHM, the mbox strips the outer IP header and trims the capability feedbacks (if any) at the packet footer. Further, the mbox needs to rewrite the ACK packet's source address back to its own address since the client's TCP connection is expecting to communicate with the mbox. Based on the received capability feedbacks, the mbox updates the *iTable* and *cTable* accordingly.

2.7.2 Capability Generation

We use the AES-128 based CBC-MAC, based on the Intel AES-NI library, to generate MAC due to its fast speed and availability at modern CPUs [28–30]. We port the capability implementation (~400 lines of `C` code) into the mbox and CHM kernel module. The mbox needs to perform both capability generation and verification whereas the CHM only performs verification.

Table 2.3: Rerouting traffic to mboxes causes small AS-hop inflation, and ~10% ASes can even access the victim with fewer hops through mboxes.

Victims	N_{infla}^{hop}	P_{cut}^{short}	P_{infla}^{no}
Non-stub ASes	1.1	10.6%	22.2%
Stub ASes	1.5	8.4%	18.0%
Overall	1.3	9.5%	20.1%

2.8 Evaluation

2.8.1 The Internet Experiments

Global Deployment. In this section, we demonstrate the feasibility of outsourcing the mboxes to the cloud via Internet-scale evaluation. Specifically, we show that rerouting clients' traffic through mboxes introduces small path length inflation and small extra latency.

We construct the AS level Internet topology based on the CAIDA AS relationships dataset [31], including 52680 ASes and their business relationships [32]. To construct the communication route, two constraints are applied [33, 34]. First, an AS prefers customer links over peer links and peer links over provider links. Second, a path is valid only if each AS providing transit is paid. Among all valid paths, an AS prefers the path with least AS hops (random tie breaker applied if necessary). We use Amazon as the cloud provider to host mboxes, and obtain its AS number based on the report [35]. Amazon claims 11 ASes in the report. We first exclude the ASes not appeared in the global routing table, and then we find AS 16509 is the provider of the rest Amazon ASes. Thus, we use AS 16509 to represent Amazon.

We randomly pick 2000 ASes as victims, and for each victim we randomly pick 1500 access ASes. Among all victims, 1000 victims are stub ASes without direct customers and the remaining victims are non-stub ASes. For each access AS and victim pair, we obtain the direct route from the access AS to the victim, and the rerouted route through the mboxes. Table 2.3 summarizes the route comparison. N_{infla}^{hop} is the average number of hop inflation of the rerouted path compared with the direct route. P_{cut}^{short} is the percentage of ASes that can access the victim with fewer hops after rerouting and P_{infla}^{no} is percentage of ASes without hop inflation.

Overall, it takes an access AS 1.3 more hops to reach the victim after rerout-

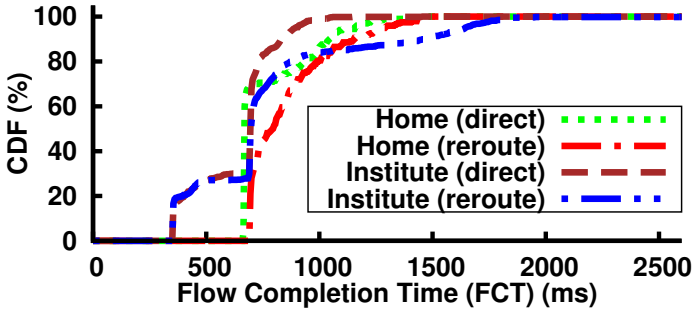


Figure 2.3: [Internet] FCTs for direct routes and rerouted routes under various Internet conditions.

ing. Even for stub victims, which are closer the Internet edge, the average hop inflation is only 1.5. We also notice that ~10% ASes in fact have shortcuts due to the rerouting.

Prototype Deployed in the Internet. In our prototype running on the Internet, we deploy three mboxes on Amazon EC2 (located in North America, Asia and Europe), one victim server in a U.S. university and about one hundred senders (located in North America, Asia and Europe) on PlanetLab [36] nodes. We also deploy few clients on personal computers to test MiddlePolice in home network. The wide distribution of clients allows us to evaluate MiddlePolice on various Internet links. Note that we cannot launch DDoS attacks over the Internet, which raises ethical and legal concerns. Instead, we evaluate how MiddlePolice may affect the clients in normal Internet without attacks, and perform the experiments involving large scale DDoS attacks on our private testbed and simulations.

In the experiment, each client posts a 100 KB file to the server, and its traffic is rerouted to the nearest mbox before reaching the server. We repeat the posting on each client for 10 thousand times to reduce bias. We also run the experiment during both the rush hour and midnight (based on the server's timezone) to further test various network conditions. As a control group, clients post their files directly to the server.

Figure 2.3 shows the CDF of the flow completion times (FCTs) for the file posting. Overall, we notice ~9% average FCT inflation, and less than 5% inflation in home network. Thus, MiddlePolice introduces small extra latency to the clients.

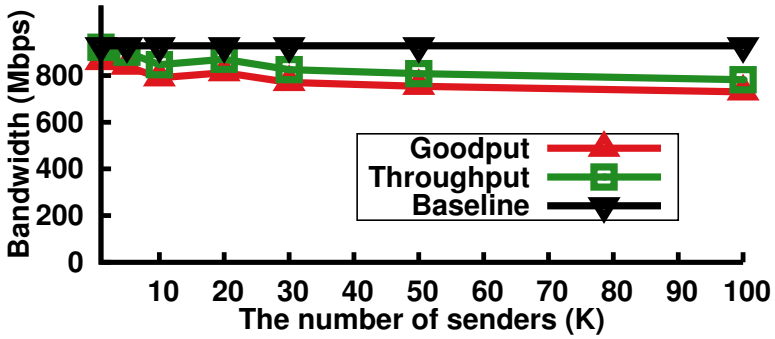


Figure 2.4: [Testbed] Throughput and goodput when policing different numbers of senders.

2.8.2 Testbed Experiments

Policing Overhead. In this section, we evaluate the traffic policing overhead on our testbed. We organize three servers as one sender, one mbox and one receiver. All servers, shipped with a quad-core Intel 2.8 GHz CPU, run the 3.13.0 Linux kernel. The mbox is installed with multiple Gigabit NICs to connect both the sender and receiver. A long TCP flow is established between the sender and receiver, via the mbox, to measure the throughput. To emulate large numbers of sources, the mbox creates an *iTable* with N entries. Each packet from the sender triggers a table look up for a random entry. We implement a two-level hash table in the kernel space to reduce the look up latency. Then the mbox generates a capability based on the obtained entry.

Figure 2.4 shows the measured throughput and goodput under various N. The goodput is computed by subtracting the additional header and capability size from the total packet size. The baseline throughput is obtained without MiddlePolice. Overall, the policing overhead in high speed network is small. When a single mbox deals with 100 thousand sources sending *simultaneously*, the throughput drops by ~10%. By replicating more mboxes, the victim can distribute the pressure of a single mbox when facing large scale attacks.

Testbed Topology. We now evaluate MiddlePolice's performance of stopping attacks. Figure 2.5 illustrates the network topology, including a single-homed victim AS purchasing one Gbps bandwidth from its ISP, an mbox and 10 access ASes. The ISP is emulated by a Pronto-3297 48-port Gigabit switch to support packet filtering. The mbox is deployed on a server installed multiple Gigabit NICs, and each access AS is deployed on a server with a single NIC. We add 100ms latency at the victim via Linux traffic control to emulate the typical In-

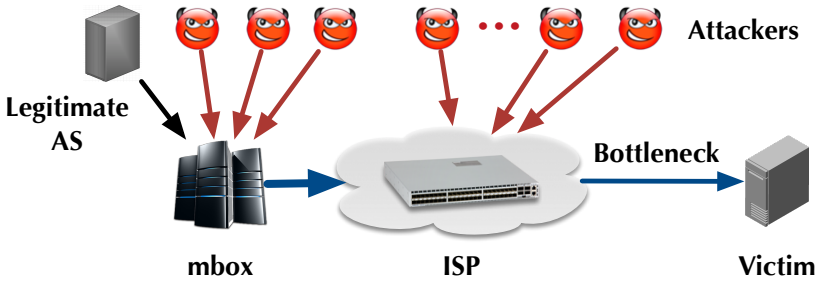


Figure 2.5: Testbed network topology.

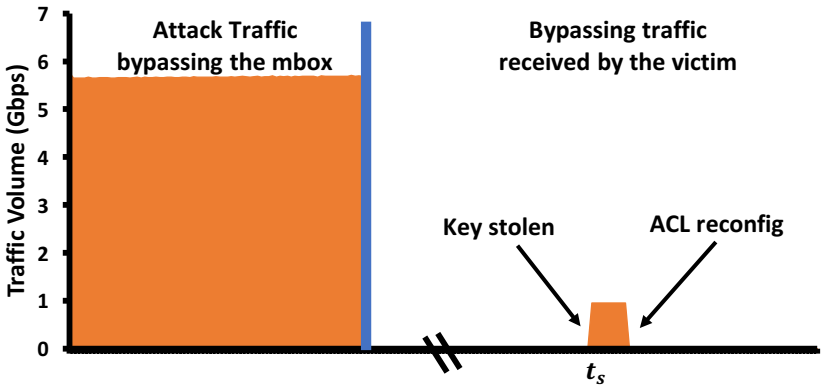


Figure 2.6: [Testbed] Packet filtering via ACL.

ternet RTT. To emulate large-scale DDoS attacks, nine ASes are compromised. Attackers adopt a hybrid attack profile: six attack ASes directly send large volumes of traffic to the victim, emulating amplification-based attacks, and the remaining attack ASes route traffic through the mbox. Thus, the total attack traffic volume is nine times as much as the client AS's rate and the victim's bottleneck capacity. Note both the inbound and outbound point of the mbox are provisioned with four Gbps bandwidth to ensure the mbox is not the bottleneck. In reality, such a property is ensured by replicating more mboxes.

Packet Filtering. We first show the effectiveness of packet filtering. Six attack ASes spoof the mbox's source address and keep dumping total six Gbps UDP traffic to the victim. The attack ASes scan all possible UDP port numbers to guess the shared secret. Figure 2.6 shows the volume of attack traffic bypassing the mbox and its volume received by the victim. Clearly, as the chance of a correct guess is small, the packet filtering mechanism can effectively stop the traffic bypassing the mbox. We further assume that at time t_s, the shared secret is stolen by attackers. However, as all packets traversing the mbox carry capabilities, the CHM suddenly receives huge amounts of packets without valid ca-

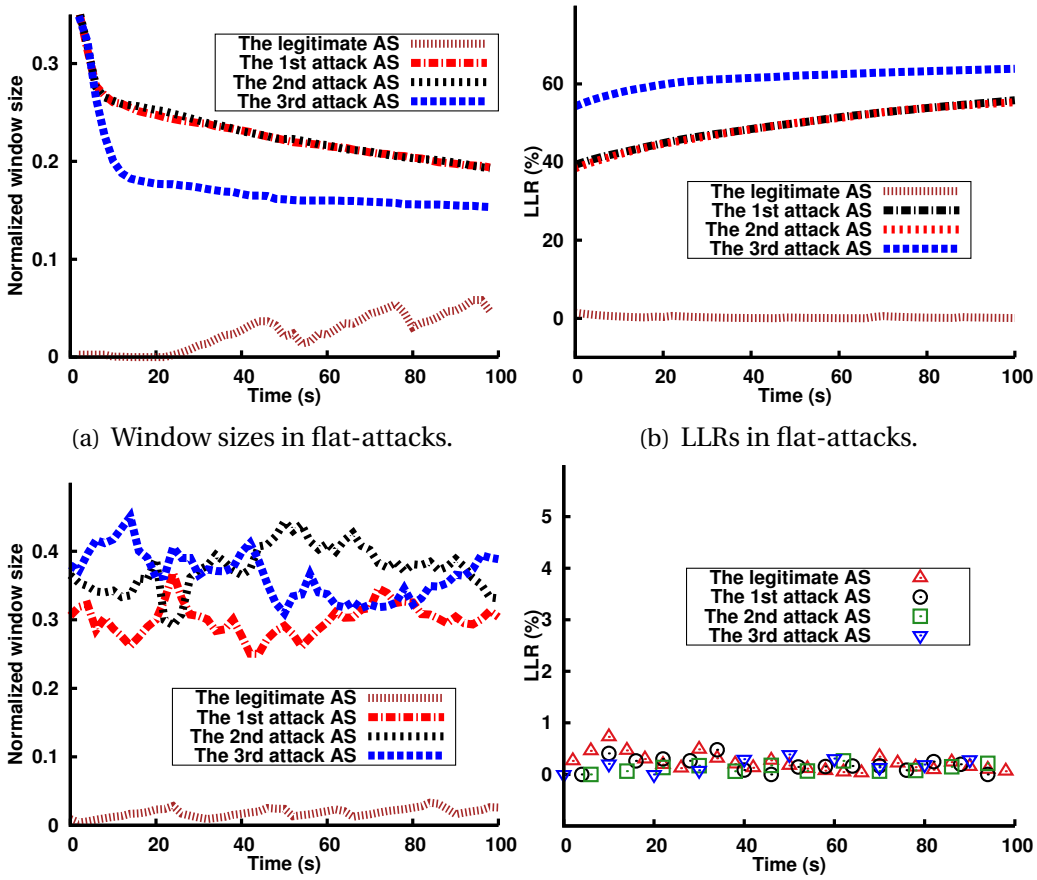
pabilities. Then it realizes that the upstream filtering has been compromised. Thus, the victim re-configures the ACL based on a new secret to recover from the key stolen. The ACL is effective within few milliseconds after reconfiguration. Thus, the packet filtering mechanism is robust even in case of secret stolen.

Bandwidth Sharing Policies. In this section, we show that the mbox can enforce victim-chosen bandwidth sharing policies. NaturalShare and PerASFairshare policies are considered. We use the default parameter setting in Table 2.2, and defer detailed parameter study in Section 2.8.3. The performance metric, defined as the window size, is the larger value between an AS's $\mathcal{W}_R$ and its delivered packets to the victim as MiddlePolice conditionally allows an AS to send faster than its $\mathcal{W}_R$. For clear presentation, we normalize the window size to the maximum number of 1.5 KB packets deliverable through a one Gbps link in one detection period. We do not translate window sizes to throughput as packet sizes vary.

Attackers adopt two representative strategies: (i) they send flat rates regardless of packet losses, and (ii) they dynamically adjust their rates based on packet losses (reactive attacks). To launch flat-rate attacks, attackers keep initiating new TCP flows regardless of whether previous flows have been finished. Thus, the attack rate is flatted by attackers' link capacity. We do not use UDP to send flat traffic since the CHM returns capabilities via TCP ACKs. Note that if a victim normally serves UDP clients, it needs to extend the CHM to return capabilities to the mboxes in other ways, *e.g.,* via dedicated flows. One way of launching reactive attacks is that attackers simultaneously maintain much more TCP flows than the legitimate AS. Such a many-to-one communication pattern may allow attackers to occupy almost the entire bottleneck in some cases, even through each single flow seems completely "legitimate".

The legitimate AS always communicates with the victim via a TCP connection.

Figure 2.7 shows the results for the NaturalShare policy. As the bottleneck is flooded by attack traffic, the legitimate AS is forced to enter timeout at the beginning, as illustrated in Figure 2.7(a). We notice that attackers' window sizes are dropping over time, which can be explained via Figure 2.7(b). Clearly, all attack ASes' LLRs are well above Th_{slr}^{drop} because their flat rates are much larger than the bottleneck's capacity. Thus, the mbox will not deliver best-effort pack-



(a) Window sizes in flat-attacks.

(b) LLRs in flat-attacks.

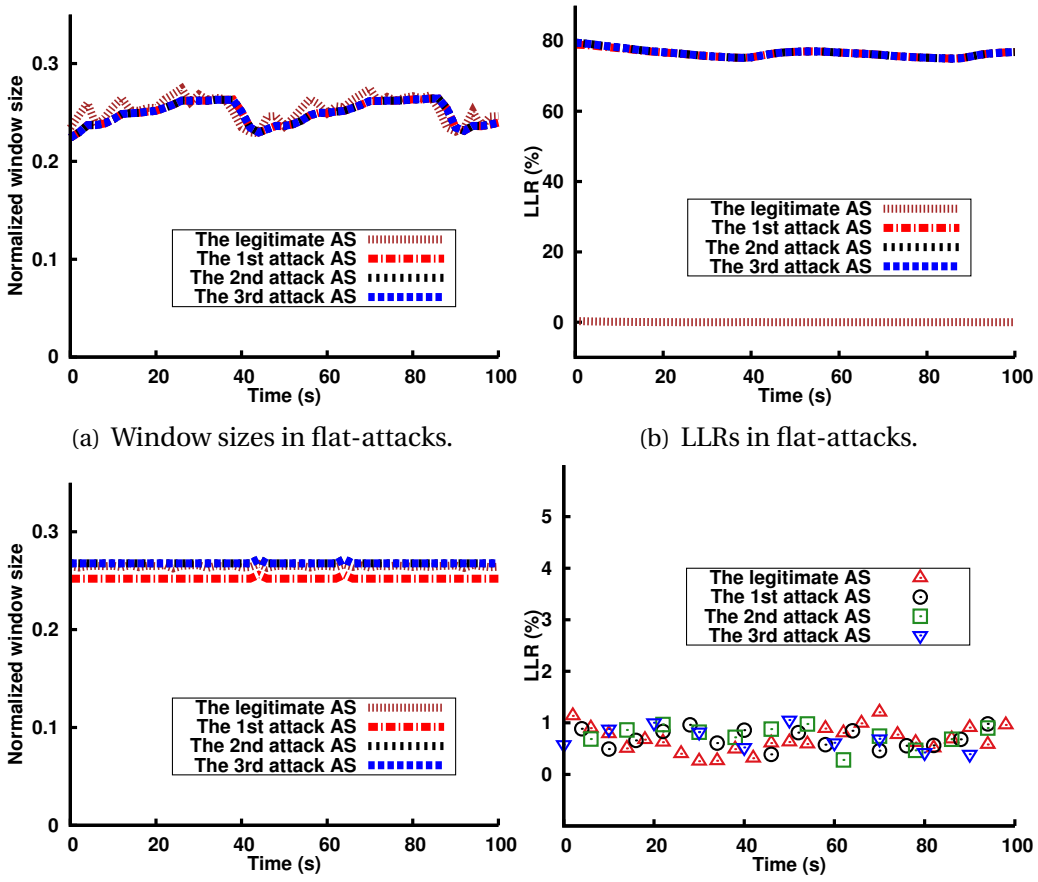
(c) Window sizes in reactive att.

(d) LLRs in reactive attacks.

Figure 2.7: [Testbed] Enforcing the NaturalShare policy. The legitimate AS gradually obtains a certain amount of bandwidth share under flat-rate attacks since attackers' window sizes consistently drop over time (Figures 2.7(a) and 2.7(b)). However, the attack ASes can consume over 95% of bottleneck bandwidth via reactive attacks (Figure 2.7(c)) while maintaining similarly low LLRs as the legitimate AS (Figure 2.7(d)).

ets for them. As a result, assuming the 1st attack AS's window size is $W(t)$ in detection period t, then $W(t+1) \leq W(t)$ no matter how many packets it sends to the mbox in period t+1. Further, any new packet losses from the attack AS, caused by either bottleneck buffer overflow due to burst or packet dropping by the victim since it sees numerous requests, will further reduce $W(t+1)$. Therefore, all attack ASes' windows are consistently dropping over time, which creates spare bandwidth at the bottleneck for the legitimate AS. As showed in Figure 2.7(a), the legitimate AS gradually recovers from the timeout.

The NaturalShare policy, however, cannot well protect the legitimate AS if attackers adopt the reactive attack strategy. By adjusting their rates based on



(a) Window sizes in flat-attacks.

(b) LLRs in flat-attacks.

(c) Window sizes in reactive att.

(d) LLRs in reactive attacks.

Figure 2.8: [Testbed] Enforcing the PerASFairshare policy. The legitimate AS can obtain at least the per-AS fair rate at the bottleneck regardless of the attack strategies (Figures 2.8(a) and 2.8(c)). Further, the legitimate AS gains a few more bandwidth than attackers under flat-rate attacks as the attack ASes have large LLRs (Figure 2.8(b)).

packet losses, attackers can keep their LLRs low enough to regain the advantages of delivering best-effort packets. Further, they can gain larger bandwidth by initiating more TCP flows. Figure 2.7(c) shows the window sizes when each attack AS starts 200 TCP long flows whereas the client AS has one. Clearly, attackers consume over 95% of the bottleneck's bandwidth, while keeping similarly low LLRs as the legitimate AS (Figure 2.7(d)).

Figure 2.8 shows the results for the PerASFairshare policy. Figures 2.8(a) and 2.8(c) demonstrate that the legitimate AS can gain at least per-AS fair rate at the bottleneck regardless of the attack strategies, overcoming the shortcomings of the NaturalShare policy. Further, under flat-rate attacks, the legitimate AS can have slightly larger window sizes than attackers since, again, the mbox does not

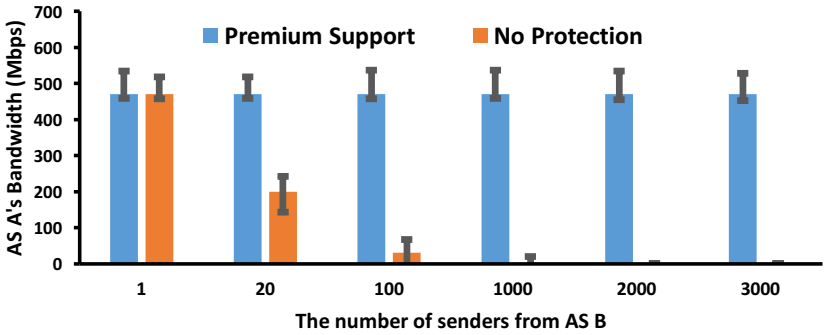


Figure 2.9: [Testbed] MiddlePolice ensures that the premium client (AS A) receives consistent bandwidth.

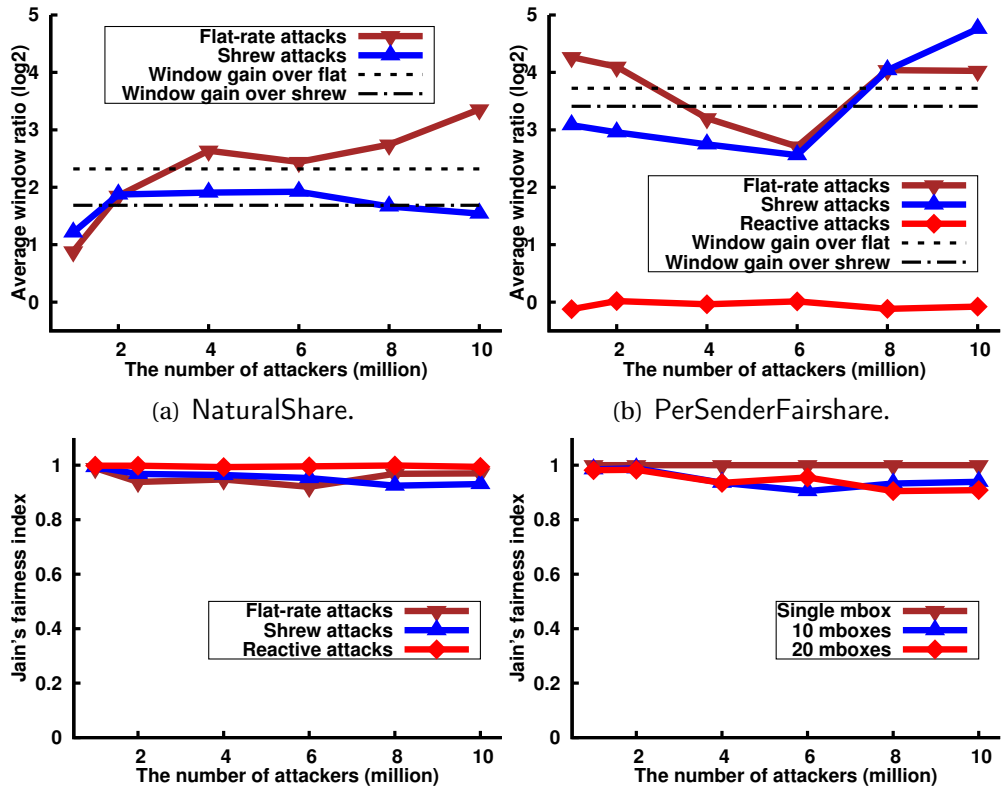
deliver any best-effort packets for attackers due to their high LLRs (showed in Figure 2.8(b)).

PremiumClientSupport Policy. This section evaluates the PremiumClientSupport policy. We consider a legitimate AS (AS A) that is a premium client which reserves half of the bottleneck bandwidth. Figure 2.9 plots AS A's bandwidth when the number of senders from the attack ASes increases. With the PremiumClientSupport policy, MiddlePolice ensures AS A receives consistent bandwidth regardless of the number of senders from the attack ASes. However, without such a policy, the attack ASes can selfishly take away the majority of bottleneck bandwidth by involving more senders.

2.8.3 Large-Scale Evaluation

In this section, we further evaluate MiddlePolice via large-scale simulations on ns-3 [37]. We desire to emulate real-world DDoS attacks in which up to millions of bots flood a victim. To circumvent the scalability problem of ns-3 at such a scale, we adopt the same approach in NetFence [13], *i.e.,* by fixing the number of nodes (about five thousand) and scaling down the link capacity proportionally, we can simulate attack scenarios where one million to 10 million attackers flood a 40 Gbps link. The simulation topology is similar to the testbed topology, except that all attackers are connected to the mbox.

Besides the flat-rate attacks and reactive attacks, we also consider the on-off shrew attacks [38] in the simulations. Both the on-period and off-period in shrew attacks are 1s. The number of attackers is 10 times larger than that of legitimate clients. In flat-rate attacks and shrew attacks, the attack traffic



(a) NaturalShare. (b) PerSenderFairshare.

(c) Jain's fairness index (FI). (d) FI for various mbox counts.

Figure 2.10: [Simulation] Evaluating NaturalShare and PerSenderFairshare in Scale. Figures 2.10(a) and 2.10(b) show that the clients' average window size is larger than that of the attackers under both flat-rate and shrew attacks. Figure 2.10(c) proves that the clients' window sizes converge to fairness in the PerSenderFairshare policy. Figure 2.10(d) shows that MiddlePolice can enforce strong fairness even without coordination among the mboxes.

volume is three times larger than the victim's link capacity. In reactive attacks, each attacker opens 10 connections, whereas a client has one. The bottleneck router buffer size is determined based on [39], and the RTT is 100 ms.

NaturalShare and PerSenderFairshare in Scale. Figure 2.10 shows the results for enforcing the NaturalShare and PerSenderFairshare policies with default parameters settings. Figure 2.10(a) illustrates the ratio of clients' average window size to attackers' average window size for the NaturalShare policy. For flat-rate attacks and shrew attacks, it may be surprising that the clients' average window size is larger than that of the attackers. Detailed trace analysis shows that it is because that the window sizes of a large fraction of attackers keep dropping. As the number of attackers is much larger than the client count, the attackers' average window turns out to be smaller than that of the clients,

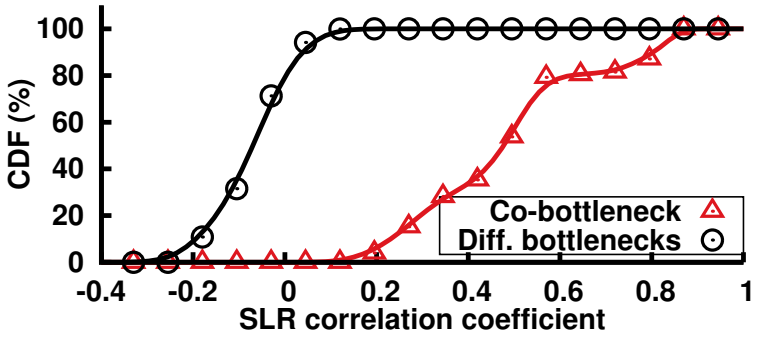


Figure 2.11: [Simulation] The SLR correlation coefficient reflects whether two mboxes share a bottleneck.

even though the absolute attack traffic volume is still higher. Under reactive attacks, the clients' average window size (almost zero) is too small to be plotted in Figure 2.10(a).

Figure 2.10(b) shows that the clients enjoy even larger window ratio gains under the PerSenderFairshare policy in flat-rate and shrew attacks because even more attackers enter the window dropping mode. Further, the PerSenderFairshare ensures that the clients' average window size is close to the per-sender fair rate in reactive attacks. Figure 2.10(c) demonstrates that each client's window size converges to the per-client fairness as the Jain's fairness index [40] (FI) is close to one.

mbox Coordination. To enforce global per-sender fairness, the mboxes sharing the same bottleneck link need to share their local observations. We first investigate how bad the FI can be without such inter-mbox coordination. We reconstruct the network topology to create multiple mboxes, and uniformly map each client to a random mbox. The attackers launch reactive attacks. The results, illustrated in Figure 2.10(d), show that the FI drops slightly by ~8% even if 20 mboxes make local rate allocations without any coordination.

To make our design complete, we also propose a co-bottleneck detection mechanism. The design rational is that if two mboxes' SLR observations are correlated, they share a bottleneck with high probability. To validate this, we rebuild the network topology to create the scenarios where two mboxes share and do not share a bottleneck, and study the correlation coefficient of their SLRs. We compute one coefficient for every 100 SLR measurements from each mbox. Figure 2.11 shows the CDF of the coefficient. Clearly, the coefficient reflects whether the two mboxes share a bottleneck. Thus, by observing such correla-

Table 2.4: [Simulation] Clients' average window size under differnent parameter settings.

(a) The NaturalShare Policy

	$\mathcal{D}_p$		Th_{slr}^{drop}		β	
	2s	8s	0.03	0.1	0.5	0.9
Flat	1.1	0.17	0.78	0.39	1.1	0.78
Shrew	1.3	0.65	0.77	1.0	1.2	0.80

(b) The PerSenderFairshare Policy

	$\mathcal{D}_p$		Th_{slr}^{drop}		β	
	2s	8s	0.03	0.1	0.5	0.9
Flat	1.0	1.1	1.0	0.69	0.85	0.81
Shrew	1.1	0.98	0.72	0.83	1.0	0.98
Reactive	1.0	0.99	1.0	0.94	1.0	1.0

tion between two mboxes' SLRs over a period of time, MiddlePolice can determine with increasing certainty whether or not they share a bottleneck, and can configure their coordination accordingly.

Parameter Study. We evaluate MiddlePolice using different parameters than the default values in Table 2.2. We mainly focus on $\mathcal{D}_p$, Th_{slr}^{drop} and β. For each parameter, we vary its value to obtain the clients' average window size under the 10-million bot attack. The results showed in Table 2.4 are normalized to the window sizes obtained with the default parameters.

Under the NaturalShare policy, the shorter $\mathcal{D}_p$ produces a larger window size. Because each sender's $\mathcal{W}_R$ is updated per-period so that a smaller $\mathcal{D}_p$ causes faster cut in attackers' window sizes. For Th_{slr}^{drop}, a smaller value slows down clients' recovery whereas a larger value allows larger window sizes for attackers. Both will reduce clients' share. A larger β has some negative effects as it takes more time for clients to recover to a low LLR.

With the PerSenderFairshare policy, MiddlePolice's performance is more consistent under different parameter settings. The most sensitive parameter is Th_{slr}^{drop} because it determines whether or not one source can send best-effort packets.

Table 2.5: Property comparison with other DDoS defense approaches. "$O(N)$ states" means that the number of states maintained by a router increases with the attacker count. "Cryptography" means that a router needs to support cryptography operation, *i.e.,* MAC computation. "Puzzle" means that the mechanism requires puzzle distribution.

	Pushback [41]	SIFF [11]	TVA [12]	Phalanx [42]
Source upgrades	**No**	Yes	Yes	Yes
Dest. upgrades	Yes	Yes	Yes	Yes
AS deployment	Unrelated	Unrelated	Unrelated	**Related**
Router support	$O(N)$ states	**None**	$O(N)$ states	**None**
Fairness regimes	None	Per-sender	None	None
Other requirements	**None**	New header	New header	New header

	NetFence [13]	Mirage [17]	SIBRA [18]	MiddlePolice
Source upgrades	Yes	Yes	Yes	**No**
Dest. upgrades	Yes	Yes	Yes	Yes
AS deployment	Unrelated	Unrelated	Unrelated	**Related**
Router support	$O(N)$ states; Cryptography	$O(N)$ states; Cryptography	$O(N)$ states; Cryptography	$O(N)$ states
Fairness regimes	Per-sender	Per-compute	Per-AS	**Victim-chosen**
Other requirements	New header; Passport [43]	Puzzle; IPv6 upgrade	Redesign the Internet	**None**

2.9 Related Work

Previous approaches can be generally categorized into capability-based approaches (SIFF [11], TVA [12], NetFence [13]), filtering-based approaches (Traceback [44, 45], AITF [46], Pushback [41, 47], StopIt [48]), overlay-based approaches (Phalanx [42], SOS [49]), deployment friendly approaches (CRAFT [15], Mirage [17]), approaches based on new Internet architectures (SCION [23], SIBRA [18], XIA [50], AIP [51]) and others (SpeakUp [19], SDN-based [52, 53], CDN-based [54]). We summarize the properties of one or two approaches from each category in Table 2.5. The comparison shows that MiddlePolice requires the *least* deployment (no source upgrades, additional router support and deployment from unrelated ASes) while providing the *strongest* property (enforcing victim-chosen policies).

2.10 Discussion

In this section, we briefly cover some undiscussed aspects.

mboxes Mapping. MiddlePolice can leverage the end-user mapping [55] to achieve better mbox assignment, such as redirecting clients to the nearest mbox, mapping clients according to their ASes and load balancing.

Other Policies. (i) Premium service. The victim needs to register premium clients' identifiers to the mboxes to enable premium service, *i.e.,* bandwidth reservation. With the source validation mechanism (detailed in Section 2.6), such identifiers can simply be the premium clients' source IPs. (ii) Per-flow fairness. Each mbox needs to maintain per-flow state and fairly allocates rates on a per-flow basis. (iii) Per-challenge fairness. Additional challenge distribution services are required to enforce this fairness metric.

Incorporating Endhost Defense. MiddlePolice can cooperate with the DDoS defense mechanism deployed, if any, on the victim. For instance, via botnet identification [56, 57], the victim can instruct the mboxes to block botnet traffic early at upstream so as to save more downstream bandwidth for clients.

2.11 Chapter Summary

This chapter presents MiddlePolice, the first readily deployable Internet DDoS defense mechanism that can enforce victim-chosen policies. MiddlePolice relies on a set of well-connected mboxes to police traffic for the victim. Attack traffic bypassing the mboxes can be effectively eliminated. Extensive evaluations on the Internet, testbed and simulations validate MiddlePolice's effectiveness for stopping attacks and enforcing policies.

CHAPTER 3

TORPOLICE: TOWARD ENFORCING SERVICE-DEFINED ACCESS POLICIES FOR ANONYMITY SYSTEMS

3.1 Introduction

Counting almost two million daily users, the Tor network [58] is among the most popular digital privacy tools. As of May 2017, the network consists of over 7,000 volunteer-run relays, carrying nearly 100 Gbps of traffic [59]. Tor clients[1] build a path (also known as circuit) consisting of three relays (guard, middle and exit) to reach service providers such as Yelp or Wikipedia. Tor is used by law enforcement, intelligence agencies, political dissidents, journalists, whistle-blowers, businesses, and ordinary citizens to enhance their online privacy [60].

Today's Tor network does not implement any access control mechanism, meaning that anyone with a Tor client can use the network without limitation. While the lack of access control fosters network growth, it has also caused various problems, most importantly botnet abuse [61]. In practice, botnets use Tor to attack third-party services, spam comment sections on websites, scrape content, and scan services for vulnerabilities [2]. In response, many service providers and content delivery networks (CDNs) have started to treat Tor users as "second-class" Web citizens [62] by forcing Tor users to solve numerous CAPTCHAs [2, 63] or blocking Tor exit relay IP addresses altogether (*e.g.*, Akamai and its powered sites).

Another type of botnet-related abuse of Tor arises from command and control (C&C) servers run as Tor hidden services [64–66]. In the past, such events caused a rapid spike in the number of Tor clients [67, 68]. Besides the reputational issue of Tor "hosting" botnet infrastructure, the massive number of circuit creation requests from botnets is a heavy burden on Tor relays, caus-

[1] In this chapter, we use the term client(s) to refer to the onion proxy (OP) software running on the Tor user's machine.

ing significant performance degradation for legitimate Tor users (*e.g.,* frequent Tor circuit failures). Other types of botnet abuse include paralyzing Tor relays via relay flooding attacks [69, 70] and performing large-scale traffic analysis via throughput or congestion fingerprinting [71, 72].

We present TorPolice, the first privacy-preserving access control framework for Tor. Leveraging cryptographically computed *network capabilities*, TorPolice allows service providers to define access policies for Tor connections, to throttle abuse while still serving legitimate Tor users. In the past, abuse-plagued service providers opted to simply block Tor users. We seek to offer a viable alternative to these service providers. Further, TorPolice improves the Tor network's resilience to botnet abuse by enabling global access control for Tor relays. Crucially, TorPolice achieves these benefits while still retaining Tor's anonymity guarantees.

TorPolice's design introduces a set of fully distributed and partially-trusted *access authorities* (AAs) that manage and certify network capabilities. Both service providers and the Tor network provide differentiated service to Tor clients that possess valid capabilities. Our AAs generate capabilities using blind signatures [73] to preserve the anonymity of Tor users.

We conduct a rigorous security analysis to prove that TorPolice completely preserves Tor's anonymity. Specifically, none of the capability-issuing authorities, service providers, or Tor relays gain any advantage in terms of linking Tor users to their online activities. We further implement a prototype of TorPolice to demonstrate its practicality and evaluate the prototype extensively to validate TorPolice's design goals. Our results show that TorPolice can effectively enforce service-selected access policies and mitigate large-scale botnet abuses against Tor at the cost of negligible overhead.

3.2 Problem Formulation

In this section, we provide brief background on Tor (Section 3.2.1), outline TorPolice's design goals (Section 3.2.2), and discuss our threat model (Section 3.2.4).

3.2.1 Tor Background

Tor clients anonymously connect to service providers by building three-hop circuits consisting of a guard, middle, and exit relay. Tor's use of layered encryption ensures that each relay only knows the identities of its direct neighbors (*i.e.*, the previous and next hop in the circuit). A list of all Tor relays—the network consensus—is published hourly by a set of nine globally distributed directory authorities that are run by volunteers trusted by the Tor Project. Clients randomly select these relays, weighted by the relays' bandwidth.[2] While the directive authorities and guard relays learn a Tor client's network identity (*i.e.*, her IP address), they cannot observe the client's online activity. Exit relays, however, can monitor the client's activity, but do not know her identity. Tor's anonymity stems from unlinking network identity from activity.

Another functionality of Tor is server-side anonymity. Tor allows service providers to host their service anonymously over hidden services (HS). Once a hidden service is set up, it creates circuits to at least three relays that are its *introduction points* (IPs). Then, the HS publishes its *descriptor*—which contains the IPs—to a distributed hash table that consists of a subset of all Tor relays. To connect to the HS, a Tor client first fetches the HS's descriptor using its *onion address*, and then builds two circuits; one to an IP and another one to a randomly-selected relay called the *rendezvous point* (RP). The client instructs the IP to send the identity of the RP to the HS, which then creates a circuit to the RP to be able to finally communicate with the client.

3.2.2 Design Goals

TorPolice adds access control to the Tor network, benefiting both service providers and the Tor network. The main challenge is to design TorPolice in a way that it (i) keeps Tor's anonymity guarantees, (ii) prevents central points of control, and (iii) is incrementally deployable.

Service-Defined Access Policies. Project Honey Pot lists nearly 70% of all Tor exit relays as comment spammers [2], causing many service providers and CDNs to block and filter traffic originating from the Tor network. To reduce this

[2]The published bandwidth of each relay in the network consensus is based on a relay's advertised bandwidth and adjusted by live bandwidth measurements that are performed by Tor's bandwidth authorities.

tension between Tor users and service providers, TorPolice must allow service providers to express access rules for Tor connections, allowing them to throttle abuse while still serving legitimate Tor users.

Mitigate Botnet Abuse Against Tor. Being a service provider itself, Tor is subject to botnet abuse, such C&C servers hosted as hidden services and (D)DoS attacks against relays. TorPolice allows the Tor network to control the network usage of Tor clients, making it possible to throttle the abuse. TorPolice's access control mechanism is *global*, meaning that an adversary cannot circumvent our defense by enumerating all relays.

Preserving Tor User Privacy. TorPolice must not degrade Tor's anonymity guarantees. While we add a new layer of functionality to Tor (access control), this layer—like Tor itself—unlinks a client's identity from her activity, and therefore preserves Tor's anonymity.

Fully Distributed and Partially Trusted Authorities. In accordance with Tor's design philosophy of distributing trust, TorPolice relies on a set of fully distributed and partially trusted access authorities (AAs) to issue capabilities. An AA is operated either by the Tor Project, a service provider, or a trusted third party. Since Tor clients are free to choose any AA to request capabilities, no single AA has a global view on all Tor clients. Further, each AA is only partially trusted and a service provider can blacklist any misbehaving AA.

Incrementally Deployable. TorPolice must be incrementally deployable. Up-to-date Tor clients, relays, and service providers can benefit from a partially-deployed TorPolice immediately while outdated entities can continue their operations.

3.2.3 Elided Design Goals

Various attacks seek to break Tor's unlinkability. For instance, an AS-level adversary may de-anonymize a Tor user's Internet activities if the adversary is in a position to monitor both ingress and egress traffic [74]. *TorPolice is not designed to mitigate those attacks on unlinkability.* Instead, we preserve the unlinkability guarantees that Tor currently provides.

3.2.4 Adversary Model and Assumptions

We consider a Byzantine adversary that deviates from our protocol and abuses Tor in arbitrary ways. The adversary can use Tor to abuse third-party services, *e.g.,* by scraping content, spamming comments, and scanning for vulnerabilities. The adversary may also abuse the Tor network directly, *e.g.,* by using Tor HSes as C&C servers, performing traffic analysis, or launching (D)DoS attacks against Tor relays. The adversary may further control a large number of bots, and hence a significant amount of resources. The bots can act passively (*e.g.,* monitor Tor traffic) or actively (*e.g.,* spoof and manipulate packets). We assume that the AAs are well-connected to the Internet backbone so that traffic-based DDoS attacks can be mitigated. Tor's existing directory authorities are subject to the same assumption.

3.3 Design Overview

In a nutshell, TorPolice is a generic access control framework based on *capabilities*. Clients must first request these capabilities, and can then spend them in return of service. TorPolice enables both service providers and the Tor network itself to enforce access control on Tor clients to mitigate various types of botnet abuse. To this end, we consider two types of capabilities: *site-specific capabilities* for accessing TorPolice-enhanced service providers through Tor, and *relay-specific capabilities* for creating TorPolice-enhanced Tor circuits. Both types of capability are signed by a set of fully distributed Access Authorities (AAs) that are deployed either by the Tor Project, service providers, or trusted third parties. To request capabilities from a particular AA, a Tor client is required to possess a *capability seed*—basically a costly-to-scale resource—accepted by the AA. Each AA accepts only a single type of capability seed. Since Tor clients are free to choose their AAs, no single AA has a global view on all Tor clients. TorPolice employs blind signatures [73] to unlink the requesting and spending of capabilities. When requesting capabilities from an AA, Tor clients express what kind of capability they request because the issuing process for two capability types differs. There are separate signing keys and rate limiters.

Figure 3.1 illustrates the requesting and spending process. While both capability types have in common step one and two, the subsequent steps differ.

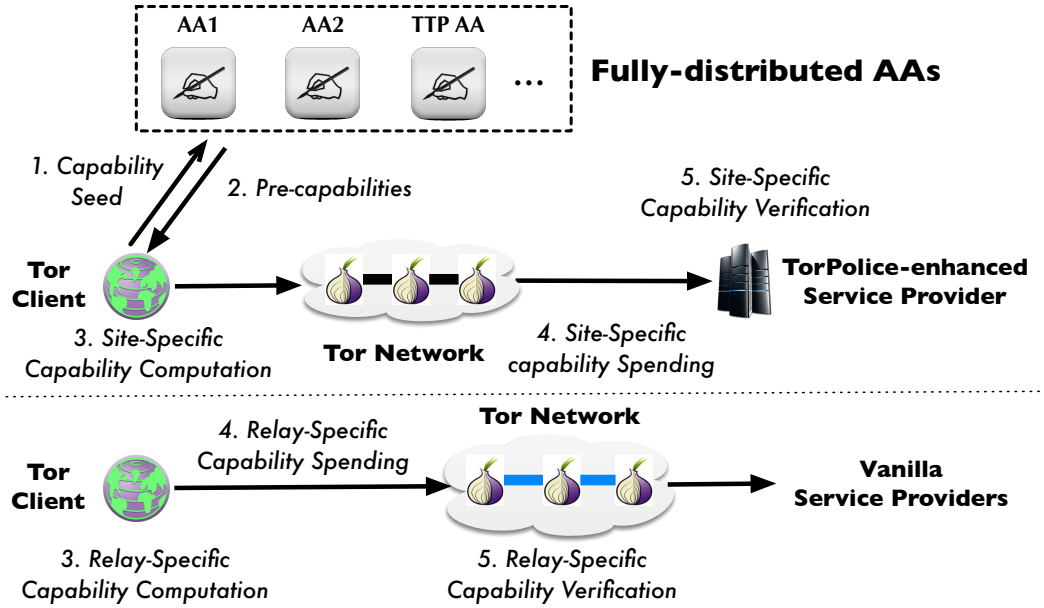


Figure 3.1: The architecture of TorPolice. A Tor client (step one) presents its capability seed to an AA to request pre-capabilities (step two), based on which the client computes site(relay)-specific capabilities (step three). The client then spends capabilities on either service access or Tor circuit creation (step four). The capability recipients validate capabilities before allowing services (step five). We intensionally separate two capability use cases for clear presentation.

A site-specific capability can only be spent at the service provider specified in the capability while a relay-specific capability is spent at a specific Tor relay to build a TorPolice-enhanced circuit. Both capability types can further be combined, *i.e.,* a Tor client can use a TorPolice-enhanced circuit to reach a TorPolice-enhanced service provider.

3.4 The Access Authorities

TorPolice relies on fully distributed and partially trusted access authorities (AAs) to manage network capabilities. We assume AAs to be honest-but-curious, meaning that they follow protocol, but seek to derive additional information about clients. An AA can be deployed by the Tor Project, service providers (*e.g.,* large CDNs like Cloudflare), or third parties. Each AA is a conceptually centralized entity. However, an AA can *distribute* its operations among multiple servers.

3.4.1 Capability Seeds

AAs expect valid *capability seeds* from Tor clients to issue *pre-capabilities*, which are the basis for deriving spendable capabilities. Any resource that is readily available to Tor users, but costly to scale, can be adopted as capability seeds. Reasonable choices include proof-of-work schemes (*e.g.,* solutions to CAPTCHAs or computational puzzles) and anonymous monetary resources. *TorPolice does not assume that capability seeds can distinguish bots from humans.* Rather, botnets can still obtain more capability seeds than legitimate Tor users. Instead, TorPolice employs capability seeds as a form of virtual identities that enable service providers and the Tor network to control resource usage by each Tor client.

In this thesis, we elaborate on two types of seeds (solutions to CAPTCHAs and computational puzzles) and further discuss how TorPolice can incorporate more types of seeds in Section 3.4.4. One key challenge of using anonymous capability seeds is to ensure that clients do not have to solve endless challenges while browsing the web and meanwhile ensure their activities are unlinkable. TorPolice proposes a capability renewal protocol to address this problem (Section 3.5.1).

Although CAPTCHAs can be deployed using publicly available libraries like reCAPTCHAs [75], TorPolice needs additional components to support computational puzzles. At a very high level, TorPolice's puzzle system design is similar to Portcullis [16]. However, TorPolice's puzzle system does make a great improvement over Portcullis: it can explicitly bound the percentage of CPU cycles that any client can spend on solving puzzles. As a result, the puzzle system can bring all bots down to the percentage that normal clients prefer to use for puzzle computation, which significantly reduce the computation disparity between the normal clients and bots. For better readability, we defer detailed design for TorPolice's puzzle system in Section A.1.

3.4.2 Per-Seed Rate Limiting

Each AA can only accept one type of capability seed. The rate at which a seed can request pre-capabilities is limited. In particular, an AA publishes two rate limiters: one determines the maximum rate at which a capability seed can request pre-capabilities used for accessing TorPolice-enhanced service providers

and the other one determines the maximum rate at which a seed can request pre-capabilities used for TorPolice-enhanced circuit creation. Based on these per-seed rate limiters published by all AAs, both service providers and Tor can configure a set of rules to fulfill their access policies. This chapter presents two concrete examples. In Section 3.5.3, we elaborate on a design that enables a site to bound an adversary's achievable service request rate through Tor using self-defined parameters. In Section 3.6.1, we present a design that allows Tor to prevent botnets from creating numerous Tor circuits to conduct various abuses. To improve readability, detailed settings of these rate limiters will be discussed when presenting these access policies.

3.4.3 Key Management

Each AA maintains two pairs of keys for signing pre-capabilities, and each of them is dedicated for one capability type. Each AA must publish the public key of both key pairs, for instance, via Tor network consensuses to ensure other entities can verify the AA's signatures. An AA can periodically renew its keys, but at any time only two key pairs from the AA are valid. After receiving signed pre-capabilities from an AA, clients must verify that the AA uses proper keys before using the pre-capabilities for accessing service providers or Tor. This prevents a malicious AA from using more keys simultaneously to partition the anonymous set. Finally, each AA is associated with a long-term fingerprint to uniquely identity the AA, similar to the fingerprint of a Tor relay.

3.4.4 Extending the Access Authorities

Besides Tor, service providers (*e.g.,* Cloudflare or Akamai) also have direct incentives to deploy and control their own set of access authorities to mitigate Tor-emitted abuses while serving anonymous connections. In fact, at the time of writing, Cloudflare is working on an independent implementation of a system whose design goals are similar to our AAs.

Finally, semi-trusted third parties such as social network operators (*e.g.,* Facebook, Google, and Twitter), may also run access authorities (shown as *TTP AA* in Figure 3.1) based on pre-agreed terms. To prevent account information leakage to Tor and service providers, Tor users only authenticate themselves to

the social network operators. Service providers or Tor only learn a single bit of information: whether a Tor client has a valid account (*i.e.,* capability seed) or not.

3.5 TorPolice-Enhanced Site Access

We now elaborate on the capability design for accessing TorPolice-enhanced service providers such as websites. To mitigate the tension between service providers and Tor users, our key observation is that service providers should not treat all connections from one Tor exit relay equally since each exit relay is shared by many Tor users. Instead, accountability should be enforced at the granularity of Tor clients, so each service provider can throttle malicious Tor clients without blocking legitimate Tor users. To this end, TorPolice designs site-specific capabilities that allows a service provider to enforce self-selected access rules on anonymous Tor connections.

3.5.1 Pre-Capability Design

Before visiting a TorPolice-enhanced site, a Tor client must first request pre-capabilities from an AA. The client is free to choose any AA based on what capability seed the client prefers to give. To request a pre-capability, the client (i) provides a valid capability seed to its selected AA and (ii) provides blinded information for the AA to compute pre-capabilities. The client can hide its network identity from the AA, for instance, by using Tor.

Capability Seed Validation. Depending on the accepted type of capability seed, an AA performs corresponding seed verification. For instance, if an AA accepts proof-of-work schemes, it needs to verify that solutions to the presented challenge are correct. Further, an AA needs to ensure that the pre-capability request rates by any seed does not exceed the two rate limiters discussed in Section 3.4.2. Since each AA maintains separate rate limiters and signing keys for two pre-capability types (*i.e.,* either for TorPolice-enhanced service access or for TorPolice-enhanced Tor circuit creation), Tor clients must specify the pre-capability type in their requests. (in this section, it is for accessing service providers).

Information Required to Compute Pre-Capabilities. To request pre-capabilities, the client provides its selected AA the following information $\{\mathbb{S}, n, \mathcal{T}_s, \mathcal{F}\}$, where $\mathbb{S}$ is the domain name of site that the client is going to visit, n is a 128-bit cryptographic nonce generated by the client, $\mathcal{T}_s$ is a universally agreed timestamp to indicate the freshness of the information and $\mathcal{F}$ is fingerprint of the selected AA. All information is blinded [73] by the client to avoid information leakage to the selected AA.

The set of information is designed to prevent abuse. In particular, $\mathbb{S}$ is used to make the capability site-specific to prevent capability double-spending at different sites. The nonce n is added to ensure the uniqueness of each pre-capability, which in turn ensures the uniqueness of each capability. The $\mathcal{T}_s$ indicates the freshness of pre-capabilities so that expired ones are nullified automatically. The client is required to use Tor's daily generated fresh random number [76] as $\mathcal{T}_s$ such that all valid capabilities have the exactly same timestamp at any time. This design eliminates the possibility of information leakage cased by timestamp abuse. $\mathcal{F}$ is added to allow other entities (*i.e.,* clients, Tor relays and service providers) to use correct public keys to verify signatures.

Computation. Upon validation of the client's capability seed, the AA computes pre-capabilities using the blinded information provided by the client. Pre-capabilities computed by an AA $\mathcal{A}_i$ are denoted by $\mathcal{P}_{\mathcal{A}_i}$. Then we have

$$\mathcal{P}_{\mathcal{A}_i} = \{\mathbb{S} \mid n \mid \mathcal{T}_s \mid \mathcal{F}_{\mathcal{A}_i}\}^b \mid \mathcal{S}^b_{\mathcal{A}_i}, \tag{3.1}$$

where $\mathcal{F}_{\mathcal{A}_i}$ is $\mathcal{A}_i$'s fingerprint, $\mathcal{S}^b_{\mathcal{A}_i}$ is $\mathcal{A}_i$'s blind signature over the set of blinded information $\{\mathbb{S} \mid n \mid \mathcal{T}_s \mid \mathcal{F}_{\mathcal{A}_i}\}^b$, and $\mid$ represents concatenation throughout this chapter.

Pre-Capability Renewal. One key design challenge for pre-capabilities is to ensure that Tor clients do not have to repeatedly solve challenges when browsing the web. A strawman design is that an AA can issue many (*i.e.,* a few hundred) pre-capabilities for each solved challenge. However, this strawman design has at least two shortcomings: (i) it breaks the site-specific pre-capability design since the client may not be able to forecast the sites that it is going to visit so as to provide these blinded information immediately after solving challenges; (ii) the design makes it easier for automated bots to accumulate pre-capabilities, weakening the entire system.

Thus, we propose a pre-capability renewal protocol to combat these prob-

lems. In particular, when a client first presents its challenge solution to an AA, the AA issues the client a unforgeable *pseudonym* as $\mathcal{I} = \{r \mid \phi\}$ where r is a random 128-bit nonce and ϕ is the AA's signature over r. Later on, the client presents $\mathcal{I}$ as a proof of validation when requesting new pre-capabilities from the AA, allowing the client to bypass future challenges. With this design, not only the site-specific pre-capability design holds, but also the AA can account each pre-capability request on a specific solved challenge (*i.e.,* capability seed) to enforce the per-seed rate limiting in Section 3.4.2. Each pseudonym has a validation period determined by the AA. Clients with expired pseudonyms are required to solve new challenges to obtain new pseudonyms that are unlinkable to previous ones.

Impact of the Pseudonym on Anonymity. Different from the pseudonym-based anonymous blacklisting systems [77, 78], in which a user interacts with a service provider using a persistent pseudonym, the pseudonym in our pre-capability renewal protocol is transient and never presented to either service providers or Tor relays. The pseudonym in our protocol is only linked with a specific challenge solution served as an anonymous capability seed. Since a Tor client presents its pseudonym to an AA through Tor, the AA cannot link the pseudonym with the client. Further, since all site-related information sent to the AA is blinded, the pseudonym is unlinkable with any site access as well. Thus, using pseudonym in our protocol does not impact Tor users' anonymity.

3.5.2 Site-Specific Capability Design

After obtaining $\mathcal{P}_{\mathcal{A}_i}$, the client *unblinds* the signature using its secret blind factor to produce the unblinded version of the pre-capability, which is the capability spendable at a specific site. In particular,

$$\mathcal{C} = \mathbb{S} \mid n \mid \mathcal{T}_s \mid \mathcal{F}_{\mathcal{A}_i} \mid \mathcal{S}_{\mathcal{A}_i} \tag{3.2}$$

The capability $\mathcal{C}$ contains a set of unblinded information that allows the site $\mathbb{S}$ to perform capability verification when the client presents $\mathcal{C}$ to access the site, as detailed in Section 3.5.3.

Employing blind signature is the key to ensure that TorPolice preserves Tor's privacy guarantee. First, signatures from the AAs prevent unauthorized entities from issuing capabilities. Second, using blind signature avoids disclosing

any site-related information to the AAs since the blinded information sent to the AAs is unlinkable with the "plain" information produced by the client. Such unlinkability further ensures the unlinkability between the client and its capability spending even if the AAs could collude with the site, which preserves online anonymity of Tor users. We provide a formal security proof in Section 3.7.

3.5.3 Site-Specific Capability Spending

Capability Validation. Tor clients spend site-specific capabilities at TorPolice-enhanced sites to request services. Upon receiving capabilities, a TorPolice-enhanced site first validates them before subsequent processing. A site-specific capability is valid if (i) it encloses an authentic signature from an AA; (ii) it encloses a domain name that is consistent with the site; (iii) the capability is not expired (*i.e.,* $\mathcal{T}_s$ is the fresh random number released by Tor); and (iv) the capability is not nullified by the site. If any of these conditions does not hold, the site rejects this capability to deny access. If a large CDN provider (*e.g.,* Cloudflare) processes capabilities on behalf of its powered sites, the second rule is passed as long as the enclosed domain is owned by one of the CDN provider's customers. In the fourth rule, whether a capability is nullified or not is determined by the site's access policies, as detailed below.

Site-Defined Access Policies. Once a site-specific capability is validated, the site accepts the Tor client's service request. Since the major form of Tor abuse is that automated bots use Tor to conduct various malicious activities against the site [2] (*e.g.,* content scraping, vulnerability scanning, comment spamming and so forth), the site needs to further control the number of service requests (*e.g.,* HTTP requests) allowed by each capability. We clarify that each site can have its own definition of service requests. Once a Tor client's service request count exceeds a threshold, the site requires a new site-specific capability for subsequent service requests. Recall that the pre-capability request rate by each client is limited by the AAs through the per-seed rate limiting design in Section 3.4.2. Thus, together with these rate limiters, it is possible for the site to design access policies so as to bound a strategic adversary's service request rate using self-selected parameters, as detailed below.

Policy Definition. Assume the following set of access authorities $\{\mathcal{A}_0, \mathcal{A}_1, ..., \mathcal{A}_n\}$ are deployed, and each authority accepts one type of capability

seed. In this context, the site defines its access policy as $\{w_0, w_1, ..., w_n\}$ where w_i is the number of service requests allowed by one valid site-specific capability issued by authority $\mathcal{A}_i$.

We now formulate $\{w_0, w_1, ..., w_n\}$ mathematically. Let $\{s_0, s_1, ..., s_n\}$ denote the set of capability seeds and authority $\mathcal{A}_j$ accepts seed s_j. Let c_j denote the cost of obtaining a capability seed s_j. We denote the cost of obtaining one network identity by λ. Let r_j denote the maximum rate at which a seed s_j can request pre-capabilities (for accessing service providers) from authority $\mathcal{A}_j$. Assume that for any client connecting to the site directly without using Tor, the site allows a maximum service request rate $\widetilde{\mathcal{O}}$ before either blocking the client or forcing the client to solve challenges. Then to bound a strategic adversary's service request rate by using Tor, the site derives $\{w_0, w_1, ..., w_n\}$ to ensure that the following condition is satisfied for any set of parameters $[\alpha_0, \alpha_1, ..., \alpha_n]$ where $\alpha_i \in [0, 1]$ and $\sum_{i=0}^{n} \alpha_i = 1$.

$$\sum_{i=0}^{n} \frac{\alpha_i \cdot \lambda}{c_i} \cdot r_i \cdot w_i \leq \epsilon \cdot \widetilde{\mathcal{O}}, \tag{3.3}$$

where ϵ is a site-defined parameter.

Policy Correctness. The parameters $[\alpha_0, \alpha_1, ..., \alpha_n]$ represent the adversary's strategy of purchasing various types of capability seeds. Thus, if Equation (3.3) holds for any strategy, the site can guarantee that the maximum Tor-emitted service request rate achieved by an adversary when spending λ on purchasing capability seeds is no greater than $\epsilon \cdot \widetilde{\mathcal{O}}$. Thus, if an adversary that spends a certain number of resources on obtaining network identities can access the site with rate $\mathcal{O}$ without using Tor, then the maximum rate that the adversary can request service from the site by using Tor is no greater than $\epsilon \cdot \mathcal{O}$, given that the adversary spends the same number of resources on acquiring capability seeds. Equivalently, in order to achieve the same service request rate, the adversary has to spend $1/\epsilon$ times as many resources when launching attacks through Tor as it spends when launching attacks natively without using Tor. To ensure that Equation (3.3) holds for any attacker strategy, we choose

$$w_i \leq \epsilon \cdot \frac{c_i \cdot \widetilde{\mathcal{O}}}{\lambda \cdot r_i}, \ \forall i \in [0, n]. \tag{3.4}$$

Policy Enforcement. If $w_i = 1$, then each capability is usable for exactly one service request. The site can enforce this by suppressing requests with dupli-

cate capabilities, for example, through the use of a Bloom filter. If $w_i > 1$, then statistically more than one service request should be allowed for each capability. To enforce this, the site stops accepting a capability with probability $1/w_i$, and then adds the capability to duplicate suppressor. However, multiple service requests carrying the same capability can trivially be linked by the site. We discuss how to address this issue through system parameterization below. Finally, if $w_i < 1$, then each capability is accepted with probability w_i, and exactly one service request is allowed for each accepted capability.

System Parameterization. We now discuss the parameterization of w_i. First, to compute w_i, the site does not need to exactly know c_i. Instead, the site simply needs to assign specific weights to these capability seeds based on its policies. Further, with an ideal parameterization, w_i should be exactly one since (i) no capability is spendable on more than one service request to ensure unlinkability and (ii) no additional capabilities are required for a single service request to avoid extra overhead. However, it is difficult to reach such ideal parameterization since r_i is chosen by the authority $\mathcal{A}_i$ that is unaware of the site's configurations ϵ and $\widetilde{\mathcal{O}}$. In addition, configurations can vary greatly among different sites so that an ideal parameterization for one site could be undesirable for others.

To address the problem, TorPolice sets r_i such that (with high probability) a Tor client can obtain enough capabilities so that it is feasible for the client to present a unique capability for each TCP connection to the site. This ensures that the client can achieve the highest level of unlinkability offered by Tor, *i.e.,* service providers only see TCP connections from Tor exit relays. We clarify that it is the client who determines how to spend its capabilities across TCP connections (as described below). The above parameterization is adopted only to ensure that spending a unique capability for each TCP connection is a feasible strategy for the client. A reasonable setting of r_i can be estimated based on the live Tor measurement in [79], which finds that in a 10-minute interval, each Tor client opens about 24 web streams. In practice, the authority $\mathcal{A}_i$ should enforce r_i over a longer period of time (*e.g.,* few hours) to accommodate usage burst.

Note that when an AA $\mathcal{A}_k$ is deployed by the site itself, system parameterization for $\mathcal{A}_k$ is easier since the site determines the rate limiters for issuing pre-capabilities.

Capability Spending. Given r_i, some sites may end up with rules $w_i > 1$, *i.e.,* one capability is allowed for multiple service requests. In this case, the site

needs to send a response to indicate whether a capability is nullified or not. Tor clients are free to determine their capability spending strategies. For instance, a Tor client can send w_i service requests using the same capability within a single TCP connection (due to HTTP keep-alive), which still ensures the highest level of unlinkability. Or the client may choose to spend one capability across multiple TCP connections to allow trans-TCP linkability. We note that if a Tor client uses the default setting of the Tor Browser, it already allows trans-TCP linkability since the Tor Browser uses session cookies. For a site that has w_i less than one, it can enforce such policies by accepting one capability with probability w_i and for each accepted capability, the site allows only one service request.

3.6 TorPolice-Enhanced Tor Access

In this section, we detail the capability design for accessing TorPolice-enhanced Tor. The current Tor network suffers from a variety of botnet abuses such as large-scale C&C abuse [61, 64–66, 68], relay flooding attacks [69, 70] and traffic analysis [71, 72]. These abuses lead to various bad results, including poor system performance for legitimate Tor users, de-anonymization threats and bad reputation for Tor. The root cause of these attacks is that botnets can create an arbitrary number of Tor circuits without any limitation. Enforcing local rate limiting for circuit creation at each relay is unlikely to stop these attacks since a strategic botnet can instruct each bot to enumerate all relays to circumvent the local rate limiting.

With TorPolice, Tor can globally control circuit creations by any client using our capability scheme. In particular, when TorPolice is activated, clients are required to possess valid capabilities in order to create TorPolice-enhanced circuits (to be incrementally deployable, circuit creation requests without valid capabilities are de-prioritized in case of congestion). Then by controlling the rate at which a Tor client can obtain capabilities, TorPolice can explicitly limit its circuit creation rate.

3.6.1 Relay-Specific Capability Design

To create a three-hop TorPolice-enhanced circuit, a Tor client $\mathbb{U}$ needs to obtain three capabilities, each of them being specific to a relay on the circuit. The

design of relay-specific capabilities is identical to that of site-specific capabilities, except for the following. (i) During pre-capability requesting, the client needs to specify the proper pre-capability type, *i.e.,* it is for Tor-enhanced circuit creations. Further, to request a pre-capability specific to a relay $\mathbb{R}$, the client encloses the fingerprint of relay $\mathbb{R}$ (rather than any site domain) in the set of blinded information sent to its selected AA. (ii) Relay-specific capabilities are spendable at TorPolice-enhanced relays for creating Tor-enhanced circuits. The relays first validate received capabilities (based on a set of rules similar to those defined in Section 3.5.3) before extending circuits.

We clarify that to request pre-capabilities, clients do not have to use TorPolice-enhanced circuits to reach the AAs. Thus, there is no deadlock for bootstrapping TorPolice. Another alternative is pre-installing few relay-specific capabilities on clients so that using TorPolice-enhanced circuits to bootstrap the system is viable.

Policy Definition. Similar to site-specific capabilities, relay-specific capabilities enable Tor to enforce access rules for its relays. In this chapter, we propose to use capabilities to limit circuit creation rate for Tor clients so as to mitigate those aforementioned abuses against Tor. In particular, assume the following set of AAs $\{\mathcal{A}_0, \mathcal{A}_1, ..., \mathcal{A}_n\}$ are deployed and authority $\mathcal{A}_i$ accepts a type of capability seed s_i. In this context, Tor defines its access rules as $\{q_0, q_1, ..., q_n\}$, where q_i is the maximum rate at which a capability seed s_i can request pre-capabilities (for creating Tor-enhanced circuits) from authority $\mathcal{A}_i$. Then in order to bound a Tor client's circuit creation rate, $\{q_0, q_1, ..., q_n\}$ should satisfy the following condition for any attacker strategy $[\alpha_0, \alpha_1, ..., \alpha_n]$ where $\alpha_i \in [0, 1]$ and $\sum_{i=0}^{n} \alpha_i = 1$.

$$\sum_{i=0}^{n} \frac{\alpha_i \cdot \lambda \cdot q_i}{3 \cdot c_i} \leq \mathcal{T}, \tag{3.5}$$

where c_i is the cost for obtaining one capability seed s_i and $\mathcal{T}$ is the maximum circuit creation rate determined by Tor. We note that the constant three appears in above equation since a standard Tor circuit contains three relays and each of them consumes a relay-specific capability.

To ensure the correctness of Equation (3.5) for any attacker strategy, we choose

$$q_i \leq \frac{3 \cdot c_i \cdot \mathcal{T}}{\lambda}, \ \forall i \in [0, n]. \tag{3.6}$$

Parameterization. Similar to how sites compute w_i in Equation (3.4), to compute q_i, the Tor project needs to assign certain weights to these capability seeds. Further, a proper configuration of $\mathcal{T}$ can be determined based on the live Tor measurements in [79]. In particular, during an average 10 minutes, PrivCount [79] estimates that a Tor client opens about four circuits. Thus, the maximum rate $\mathcal{T}$ at which one Tor client can create circuits should be close to four per ten minutes. In practice, each AA should enforce q_i over a longer period of time (*e.g.,* few hours) to accommodate usage bursts and relay churn.

3.6.2 Capability Exchange for HSes

The design of relay-specific capabilities needs to be augmented with a *capability exchange protocol* to better support Tor Hidden Services (HSes). In particular, a Tor hidden server (itself runs a Tor client) needs to open many Tor circuits in order to serve all its clients (referred to as HS-clients). Although a Tor hidden server can continue to use legacy Tor circuits to serve its HS-clients, we do design a capability exchange protocol to enable hidden servers to build enough TorPolice-enhanced circuits.

The design intuition is that a HS-client requests a new type of capability, *i.e., trans-capability*, from the AAs, and sends it to the HS, which subsequently redeems the trans-capability at the AAs for new pre-capabilities. The trans-capability, accounted on the capability seed of the HS-client, anonymously informs the AAs that the hidden server needs to create a new TorPolice-enhanced circuit to serve the HS-client. For better readability, the detailed design of the protocol is deferred in Section A.2.

3.7 Security Analysis

In this section, we analyze the impact of TorPolice on Tor users' anonymity. Let $\mathbb{N}_T$ denote the set of Tor clients that request pre-capabilities from the AAs, and subsequently present capabilities to access service providers or Tor relays. We first present lemmas on unlinkability.

3.7.1 Lemmas

Lemma 1. *Consider any client* $\mathbb{U} \in \mathbb{N}_T$. *By colluding with each other, both the AAs and a service provider* $\mathbb{W}$ *gain only negligible advantage over random guessing when trying to link a specific site access* $\mathbb{V}$ *using Tor with the client* $\mathbb{U}$.

Proof. We start the proof with the following denotations. Let $\mathbb{V}$ denote a site access to $\mathbb{W}$ initiated by the client $\mathbb{U}$ using Tor. Let $\mathcal{C}$ denote the service-specific capability that $\mathbb{U}$ sends to $\mathbb{W}$ to support the access $\mathbb{V}$. Let $\mathcal{P}$ denote the pre-capability used by $\mathbb{U}$ to compute $\mathcal{C}$.

Since the client $\mathbb{U}$ can use Tor to connect to the AAs when requesting the pre-capability $\mathcal{P}$, in the ideal case, $\mathbb{U}$ is unlinkable with $\mathcal{P}$. However, to ensure that our lemma still holds in the worst case when Tor's unlinkability is broken by adversaries, we assume the AA that issues $\mathcal{P}$ can link $\mathcal{P}$ with the client $\mathbb{U}$. Thus, the service provider $\mathbb{W}$ and other AAs can have such linkability as well by colluding with the AA.

Next, we prove the lemma by contradiction. Assume that the AAs and $\mathbb{W}$ can design an algorithm $\mathcal{K}$ that enables the AAs and $\mathbb{W}$ to link site access $\mathbb{V}$ with the client $\mathbb{U}$. Since the site access $\mathbb{V}$ is linkable with the capability $\mathcal{C}$ (as $\mathcal{C}$ is presented for the access $\mathbb{V}$) and the client $\mathbb{U}$ is linkable with the pre-capability $\mathcal{P}$ (based on the above assumption), designing algorithm $\mathcal{K}$ is equivalent to designing an algorithm $\mathcal{K}'$ that enables the AAs and $\mathbb{W}$ to link the capability $\mathcal{C}$ with the pre-capability $\mathcal{P}$.

In TorPolice's design, $\mathcal{P}$ is the blinded message signed by an AA (*i.e.,* the blind-signer), and $\mathcal{C}$ is the unblinded version of $\mathcal{P}$ produced by a secret factor unknown to the blind-signer. Thus, the problem of designing $\mathcal{K}'$ to link $\mathcal{P}$ with $\mathcal{C}$ is the same as designing an algorithm $\mathcal{K}''$ that allows a blind-signer to link the blinded message it signs to the unblinded message without knowing the secret factor, which is impossible in a blind signature [73, 80]. This contradiction proves that the hypothetical algorithm $\mathcal{K}$ does not exist, indicating both AAs and $\mathbb{W}$ gain only negligible advantages of linking $\mathbb{V}$ with $\mathbb{U}$ via collusion. We clarify this lemma does not claim that colluding among multiple entities does not pose a risk; it only proves that TorPolice does not introduce any further risk. □

Using the similar reduction proof as Lemma 1, we can prove the following lemma.

Lemma 2. *Consider any client* $\mathbb{U} \in \mathbb{N}_T$. *By colluding with each other, both the AAs and Tor relays gain only negligible advantage over random guessing when trying to link a specific relay access with* $\mathbb{U}$.

3.7.2 Information Leakage Analysis

We now analyze the information leakage to an arbitrary service provider $\mathbb{W}$ using *degree of anonymity* [81, 82]. The analysis uses information-theoretic entropy [83] as the measure of information contained in a probability distribution. Recall that $\mathbb{N}_T$ denote the set of TorPolice-enhanced clients. Given an arbitrary capability-enhanced site access, $\mathbb{W}$ believes that with probability p_i the access originates from client i in $\mathbb{N}_T$, *i.e.,* $\mathbb{W}$ maintains a probability distribution I for anonymous accesses. Then, the entropy (*i.e.,* the information contained in the distribution I) is $H_{\mathbb{W}} = -\sum_{i \in \mathbb{N}_T} p_i \cdot \log_2(p_i)$.

Based on Lemma 1, we have $p_i = \frac{1}{N_T}$, where N_T is the size of the anonymity set $\mathbb{N}_T$. Thus, the entropy of the Tor network after introducing TorPolice is $H_{\mathbb{W}} = \log_2 N_T$.

Next, we analyze the system entropy before introducing TorPolice. Let $\mathbb{N}$ denote the entire set of anonymous Tor users. Before introducing TorPolice, the maximum entropy H_M of the Tor network in which the set of anonymous users $\mathbb{N}$ may access $\mathbb{W}$ is $H_M = \log_2 N$, where N is the size of the anonymity set $\mathbb{N}$.

Thus, based on the definition in [81, 82], the degree of anonymity d after introducing TorPolice is

$$d = 1 - \frac{H_M - H_{\mathbb{W}}}{H_M} = \frac{\log_2 N_T}{\log_2 N}. \tag{3.7}$$

Since Tor clients connect to the AAs using Tor, from the perspective of the AAs and $\mathbb{W}$, the anonymous set remains the same after introducing TorPolice, *i.e.,* $\mathbb{N}_T = \mathbb{N}$. Thus, we have $d = 1$ based on Equation (3.7), proving that there is no information leakage to the service provider $\mathbb{W}$ after introducing TorPolice.

The above information leakage analysis is equally applicable to Tor relays and the AAs since same as the $\mathbb{W}$, neither of them can link a particular capability spending with a specific Tor client in $\mathbb{N}_T$ (as stated in Lemma 2).

3.8 Implementation

We now present the full implementation of TorPolice.

3.8.1 Capability Implementation

We implement capability-related computation using C, Python and JavaScript to consider various usage scenarios. For instance, the capability design can be directly built into the Tor software written in C (as shown in Section 3.8.4), or it can be implemented as a plugin for the Tor browser, which executes capability-related computation in JavaScript (as shown in Section 3.8.2). Websites may compute capabilities using any language. Thus, we use Python as an example due to its popularity in web applications.

We use the RSA algorithm to perform capability-related cryptographic operations such as blind signing. The C implementation uses the OpenSSL library [84] and the Python implementation imports the PyCrypto module [85]. Since no standardized JavaScript library for computing blind signatures is available, we develop our own library based on crypto-js [86] and BigInt [87], two libraries that allow us to perform computation (*e.g.,* modulo) for very large prime numbers in JavaScript. We benchmark the capability computation overhead in Section 3.9.1.

3.8.2 AA Implementation

For an AA accepts CAPTCHAs as capability seeds (referred to as CAA), we implement it as a web server that deploys Google's reCAPTCHA [75] service. For an AA accepts computational puzzles (referred to as PAA), it accepts puzzle solutions over HTTP requests. These AA servers define a customized HTTP header (X-Capability) to carry TorPolice-related cryptographic tokens such as pseudonyms and pre-capabilities. To make the implementation transparent to clients (*i.e.,* no client-end network stack modifications are required), the AA servers add X-Capability in the Access-Control-Allow-Headers HTTP header option.

Although the PAA can be accessed using native HTTP libraries, the CAA needs to be accessed using browsers. Thus, we implement a Firefox add-on (referred to as CapJS) to execute TorPolice-related cryptographic operations in browsers. In real-world deployment, the add-on should be developed by trusted entities

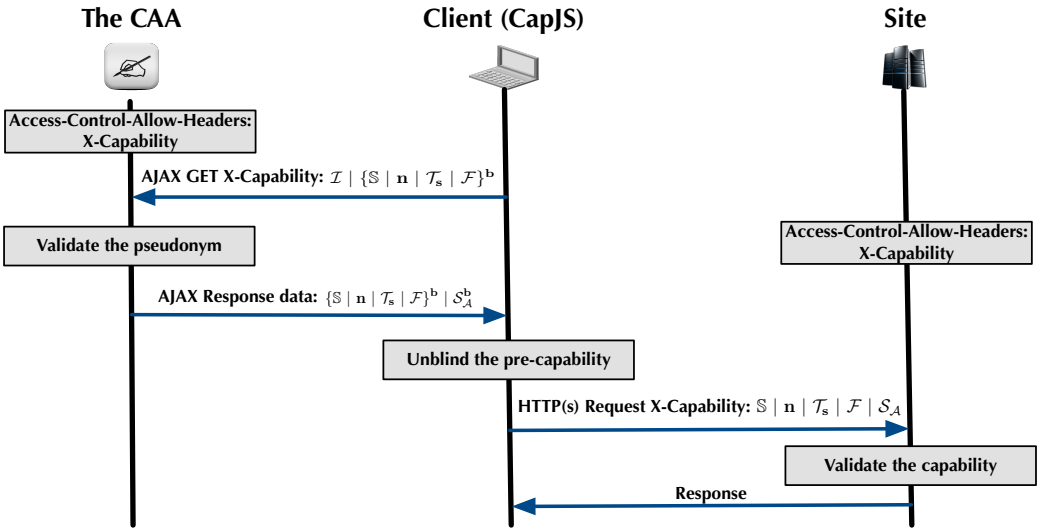


Figure 3.2: Site access by a client with CapJS installed.

(*e.g.*, the Tor project) and signed by Mozilla so that Tor users can install it on their Tor browsers.

CapJS Design. When a Tor client connects to a CAA server, CapJS checks cookies to determine whether a pseudonym $\mathbb{J}$ issued by the CAA server is locally cached. If so, CapJS loads $\{\mathbb{J} \mid \{\mathbb{S} \mid n \mid t_s\}^b\}$ into the X-Capability header, where $\{\mathbb{S} \mid n \mid t_s\}^b$ is the set of blinded information described in the pre-capability design (Section 3.5.1). If no pseudonym is available, CapJS only loads the set of blinded information into the X-Capability header. With this customized HTTP header, CapJS sends an AJAX GET to the CAA server.

When receiving the AJAX request, the CAA server inspects the X-Capability header. If a valid pseudonym is retrieved, the CAA server computes a pre-capability for the client using the blinded information carried in the header. Otherwise, the CAA server loads a reCAPTCHA challenge page for the client. Once the challenge is successfully solved, the CAA server computes a pre-capability by signing the blinded information, as well as a pseudonym for the client. These tokens are returned to the client in a JSON object responding to the client's AJAX GET request.

After receiving a response from the CAA server, CapJS inspects the received data object to retrieve the pre-capability and the pseudonym. The pre-capability is then unblinded to produce a capability, and the pseudonym is cached for future use.

CircID	CREATE / EXTEND	Capability || Original Payload

Figure 3.3: TorPolice-enhanced Tor circuit creation.

3.8.3 TorPolice-Enhanced Site Access

To serve TorPolice-enhanced Tor clients, the deployment required at websites is lightweight. In particular, a site needs to add X-Capability in its Access-Control-Allow-Headers HTTP header option to allow CapJS to pass site-specific capabilities in the header. Upon receiving capabilities, the site verifies them using the rules defined in Section 3.5.3 to fulfill its access policies. Figure 3.2 depicts a site access by a client with CapJS installed on its browser.

3.8.4 TorPolice-Enhanced Tor Circuit

We now discuss the implementation of TorPolice-enhanced Tor circuit creation.

Tor Source Code Modification. We modify the Tor software source code to integrate our capability design into Tor circuit creation. The native circuit creation proceeds as follows. The onion proxy (OP) on a Tor client sends a CREATE cell containing the first half of the DH handshake to a guard relay, which responds with a CREATED cell containing the second half of the handshake. To extend the circuit to a new relay $\mathbb{R}_e$, the OP sends a RELAY_EXTEND cell (specifying the address of $\mathbb{R}_e$ and a new secret) to the last relay $\mathbb{R}_m$ on the partially created circuit. $\mathbb{R}_m$ copies the received information into a new CREATE cell, and forwards the cell to $\mathbb{R}_e$.

To create a capability enhanced circuit, in addition to these original cells, the OP further sends a valid relay-specific capability to each hop. In our prototype, the OP *prepends* a capability to the payload of the CREATE cell when connecting to the guard relay. Capabilities for subsequent relays are prepended to corresponding RELAY_EXTEND cells. Figure 3.3 illustrates this structure. Each relay first verifies the received capability based on the rules defined in Section 3.6.1 before processing the onionskin carried in the remaining payload. Since capability verification is much cheaper than the onionskin processing, this design saves the relay considerable compute resources for processing circuit creations without valid capabilities. Alternatively, a relay-specific capability can be carried via a customized cell. In this case, the cell should be sent

together with the CREATE cell (or RELAY_EXTEND cell) to avoid additional RTTs.

To validate our implementation, we test the modified Tor source code in Shadow [88], a safe development environment to run real Tor source code in a private Tor network. Via log analysis, our test experiments show that our implementation properly embeds relay-specific capabilities into the workflow of Tor circuit creation.

Live Tor Interaction. Since live Tor relays do not run our modified Tor source code, we cannot create TorPolice-enhanced circuits directly through live Tor relays. Thus, we implement *another* prototype to interact with live Tor relays, as detailed in Section A.3.

3.9 Evaluation

In this section, we present a detailed evaluation for TorPolice to demonstrate the following.

TorPolice Introduces Small System Overhead. We show that capability-related operations introduce small overhead compared with the typical Tor circuit creation latency (Section 3.9.1). Further, we show that the deployment overhead of the AAs is small. For instance, the AAs collectively need only 11 cores to support the entire set of current Tor users (Section 3.9.2).

TorPolice Effectively Enforces Site-Defined Policies. We demonstrate that TorPolice allows a site to effectively enforce its access policies on anonymous Tor users, *i.e.,* the site can bound service request rate by any strategic adversary via self-defined parameters (Section 3.9.3).

TorPolice Mitigates Various Abuses Against Tor. Based on real data collected by Tor, we demonstrate TorPolice can mitigate large-scale C&C abuse and prevent cell flooding attacks against the Tor network (Section 3.9.4).

3.9.1 Capability Computation Overhead

In this section, we benchmark the overhead of capability-related computation in C, Python and JavaScript on our testbed. All results are obtained using a single 3.30 GHz Intel i3-3120 core. We perform 10,000 runs to learn the mean, me-

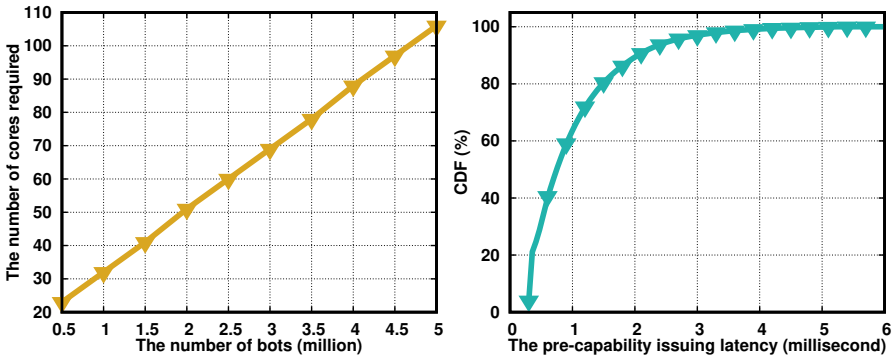
Table 3.1: The computational time (in microseconds) for capability-related operations.

Operation	Mean	Median	Std. Dev.	Language
Generation	232.0	232.0	0.1	C
	253.7	253.6	0.3	Python
	27,320.0	27,240.0	245.5	JavaScript
Verification	25.6	25.6	0.0	C
	32.0	32.0	0.1	Python
	355.5	354.3	5.3	JavaScript
Blinding	3.5	3.5	0.0	C
	46.3	46.3	0.1	Python
	18.1	18.1	0.3	JavaScript
Unblinding	2.4	2.4	0.0	C
	7.0	7.0	0.0	Python
	64.8	64.7	6.8	JavaScript

dian, and standard deviation of the computation times for a single capability generation, verification, information blinding and unblinding. Results shown in Table 3.1 are obtained when the RSA key length is 1024. The overall computational overhead is small. For instance, it takes an AA ~230 microseconds in C to compute a pre-capability. And a single capability verification takes ~25 microseconds in C. A blinding and an unblinding operation by Tor clients can be finished in about three and two microseconds, respectively, in C. The implementations in C and Python have comparable performance. Although it is more expensive to perform signing and verifying in JavaScript, the overhead of blinding and unblinding operations (performed by Tor clients) in JavaScript is comparable with other languages. The AAs, relays and service providers can adopt more efficient languages such as C and Python.

3.9.2 Deployment Overhead of the AAs

In this section, we evaluate the deployment overhead of the AAs. We first estimate the compute resources needed by the AAs to support pre-capability issuing for all Tor users. Then we evaluate the pre-capability issuing latency using the AAs deployed on our testbed.



(a) The # of cores required. (b) Pre-capability issuing latency.

Figure 3.4: Figure 3.4(a) shows the number of cores required to prevent the AAs from being flooded. Figure 3.4(b) plots pre-capability release latency benchmarked on our testbed.

Collective Compute Resources Needed. To estimate compute resources required from the AAs, we need to estimate the amount of pre-capability requests from all Tor clients. Recall that each AA server maintains two rate limiters for issuing pre-capabilities: r for issuing site-specific pre-capabilities and q for issuing relay-specific pre-capabilities (Section 3.4.2). We estimate r and q using the live Tor measurement results in [79]. In particular, during a 10-minute interval, PrivCount [79] estimates that each Tor client opens about 24 TCP streams and four circuits. Further, PrivCount [79] counts about 710,000 unique clients during a 10-minute interval. Combing these, we estimate the collective pre-capability request rate from all Tor clients is about 44,000 per second. Since it takes one core 0.23 milliseconds to issue one pre-capability, the AAs collectively need about 11 cores to support the entire set of current Tor users.

In practice, the AAs should be over-provisioned to prevent an adversary from flooding them via massive pre-capability requests. We clarify that such flooding attack aims to exhaust the AAs' compute resources rather than their network bandwidth (bandwidth-oriented DDoS attacks can be prevented by hosting the AAs on well-provisioned cloud [89]). Figure 3.4(a) plots the number of cores required in order to withstand different-sized botnets. The results show that the AAs need about 100 cores to withstand a five-million node botnet.

Pre-Capability Release Latency. We now evaluate pre-capability release latency using the AAs deployed on our testbed (Section 3.8.2). We define the pre-capability release latency as the time required for an AA server to process a pre-capability request, excluding networking latency and other user-introduced la-

tency (*e.g.,* the time required for solving challenges). We provision eight servers on our physical testbed as AA servers in this experiment. We double-threaded each AA server so that the eight AA servers collectively have 16 cores. To emulate pre-capability requests from the entire set of Tor clients, we develop a requester that generate requests at the rate of 44,000 per second. To send each request, the requester randomly picks one of the eight AA servers. The results, plotted in Figure 3.4(b), show that the pre-capability release latency is less than few milliseconds, which is over two orders of magnitude smaller than the typical Tor circuit creation time (about 0.7s based on our live Tor measurements in Section A.3).

3.9.3 Enforcing Site-Defined Policies

In this section, we demonstrate that TorPolice enables a site to enforce site-defined access policies on anonymous Tor clients: *i.e.,* a site can bound a strategic adversary's service request rate using self-defined parameters.

Access Policies. For simplicity, we assume that the site assigns equal weights to both types of capability seeds, *i.e.,* c_0 (for CAPTCHAs) and c_1 (for puzzles) in Equation (3.4) are the same. However, the actual costs, denoted by c_0' and c_1', can be different from c_0 and c_1. Further, base on the measurements in [90, 91], we assume c_0' is close to λ (the cost for obtaining one network identity).

We evaluate three strategies that a site may use to define its access policies. The first strategy (referred to as *basic strategy*) is that the site accepts all Tor requests with valid capabilities. In the second strategy (referred to as *rate limiting strategy*), the site enforces a maximum service request rate $r_{\max}$ for *all* Tor requests with valid capabilities. In the third strategy, besides rate limiting, the site further performs weighted fair queuing (WFQ) to serve requests: rather than serving all valid requests in one FIFO queue, requests with capabilities obtained using CAPTCHA solutions and puzzle solutions are served in two separate FIFO queues weighted equally. The third strategy (referred to as *WFQ strategy*) prevents one type of seed from overwhelming the other one.

Policy Enforcement. We now study an adversary's service request rate through Tor when it invests a certain amount of money on acquiring capability seeds. Define $k = c_0'/c_1'$. We first present the evaluation results for $k = 0.5$ in Figure 3.5 and then extend our discussion to arbitrary k. For any amount of investment,

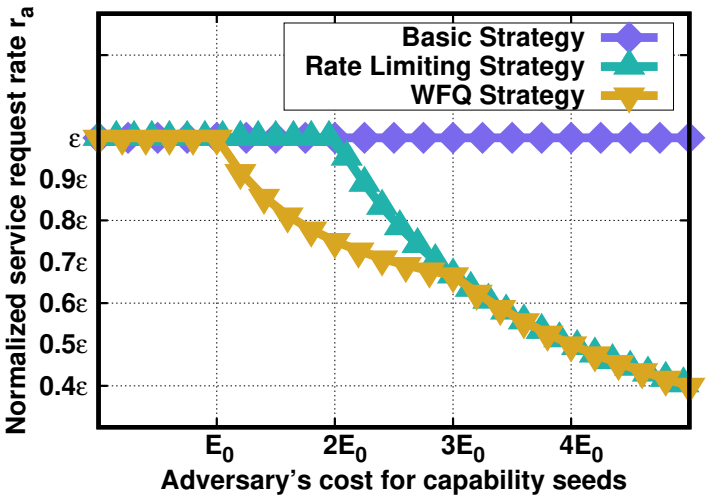


Figure 3.5: TorPolice enables a site to bound an adversary's service request rate using self-defined parameter ϵ. E_0 is the adversary's cost at the point of diminishing returns.

the adversary's service request rate through Tor (denoted by r_a) is normalized to the service request rate obtained when the adversary connects to the site directly without using Tor.

Since $c'_0 < c'_1$ given $k = 0.5$, the adversary's optimal strategy is spending all investment on CAPTCHA solving. Thus, we have $r_a = \epsilon$, where ϵ is the site-configurable parameter defined in Equation (3.3). When the site adopts the basic strategy, r_a remains the same as the adversary increases its investment. However, for the other two strategies, r_a will reach a point of *diminishing returns* as the adversary's investment further increases (as shown in Figure 3.5). In particular, when the site adopts the rate limiting strategy, the point of diminishing returns is reached when the collective service request rate from the adversary and all legitimate Tor clients exceeds $r_{\max}$. In Figure 3.5, we denote the adversary's cost at this point by $2E_0$. After that, further increasing investment actually reduces r_a since no more Tor requests are allowed by the site.

When the WFQ strategy is adopted, r_a experiences two points of diminishing returns as the adversary's cost increases, as shown in Figure 3.5. The first one happens when the collective service request rate from all Tor clients using the optimal seed (CAPTCHAs in our evaluation) exceeds $\frac{r_{\max}}{2}$. After this point, the adversary has to use sub-optimal seeds in order to further get services. As a result, r_a starts to decline from the optimal rate ϵ. The second point of diminishing returns is reached when the collective Tor-emitted service request rate

exceeds r_{max}.

General Results. Our further analysis (deferred in Section A.4) proves that for any k, $r_a \leq \epsilon$ if $k \leq 1$ and $r_a \leq k \cdot \epsilon$ if $k \geq 1$. Thus, regardless of the actual cost of obtaining capability seeds, the adversary's service request rate is bounded by $\Theta(\epsilon)$. This result holds no matter which strategy the site adopts and how many types of capability seeds are accepted by TorPolice.

3.9.4 Mitigating Abuse Against Tor

In this section, we perform Tor-scale evaluations to demonstrate the following. (i) TorPolice effectively mitigates large-scale botnet C&C abuse against Tor by reducing circuit failure ratios by ~74% (Section 3.9.4.1) and (ii) TorPolice significantly increases Tor's resilience against cell flooding attacks that aim to paralyze Tor via excessive circuit creations (Section 3.9.4.2).

Tor-Scale Simulator. We aim to show that TorPolice is able to mitigate the harm that a multi-million botnet can do on Tor. While we do have a TorPolice implementation that runs on Shadow [88] (see Section 3.8.4), we would run into scalability issues with simulating millions of Tor clients. Further, Shadow is unable to help us simulate the cryptographic overhead that botnets would impose on Tor relays [92]. Due to these shortcomings, we developed our own simulator. We faithfully implement Tor's path selection algorithm [93] and validate the correctness of our implementation by comparing relays' selection probability with the ones published by Tor Atlas [94]. We sampled the computational capacity of relays from Barbera *et al.*'s work that was based on live Tor measurements [69].

3.9.4.1 Mitigating Botnet C&C Abuse

In this section, we study the botnet C&C abuse that happened during Aug-Sep 2013, when Tor's daily estimated users rapidly increased from one million to six million. We show that Tor clients experienced very high circuit creation failure rates when Tor was under this abuse. Then we show TorPolice effectively mitigates such abuse.

Circuit Creation Failure Rate. We use the data collected by Tor to estimate the amounts of circuit creations initiated by the botnet during the C&C abuse.

To improve readability, we defer the detailed modeling to Section A.5. Due to the massive circuit creations by the botnet, compute resources of many relays are exhausted, resulting in very high circuit creation failure rates, as depicted in Figure 3.6. Such high failure rates are caused by the following vicious cycle. When the abuse starts, Tor relays begin to drop requests due to the lack of compute resources. These initial failures force the bot clients to continuously send requests until their circuits are successfully created, which further increases the network load. The resulting consequences are that the botnet still managed to use Tor as its primary C&C channel after numerous trials whereas Tor is less usable for legitimate Tor users since it could require tens of trials before a circuit is finally created, resulting a high user-perceived latency.

Mitigating Botnet Abuse in Tor. The root cause of such high circuit creation failure rates is that bot clients deviate from typical Tor usage pattern, *i.e.,* they initiate numerous circuit creation requests without any limitation. As described in Section 3.6.1, TorPolice allows Tor to explicitly control the circuit creation by Tor clients. Thus, to counter this abuse, TorPolice sets its access policies q_i so that the maximum rate at which a client can create circuits is four per ten minutes (aligned with live Tor measurements in [79]). We plot the resulting circuit creation failure rates after enforcing the access policies in Figure 3.6. Clearly, TorPolice effectively eases the network load and reduces the average failure rate from ~41% to ~10%, a ~74% reduction.

In response to the C&C abuse, the Tor project released a new version (0.2.4.17-rc) that prioritizes the processing of onionskins using the `tnor` [95] protocol since the bot clients used an older version without `tnor` support. Tor's countermeasure reduced the circuit failure ratio to about 20% [68]. We clarify that a strategic botnet could circumvent Tor's defense by changing adaptively (*e.g.,* upgrading software). However, TorPolice offers long-term countermeasures that can handle strategic botnets.

3.9.4.2 Mitigating Tor-Targeted DDoS Attacks

As noted in [61], a general concern of attacking a botnet by Tor in case of abuse (*e.g.,* by blacklisting its hidden servers) is that it may lead to retaliation. For instance, a botnet can easily paralyze Tor via excessive circuit creation requests. According to Tor design [93], a Tor client drops its current guard relay when circuit failure rate measured by the client is above 30%. Via massive circuit

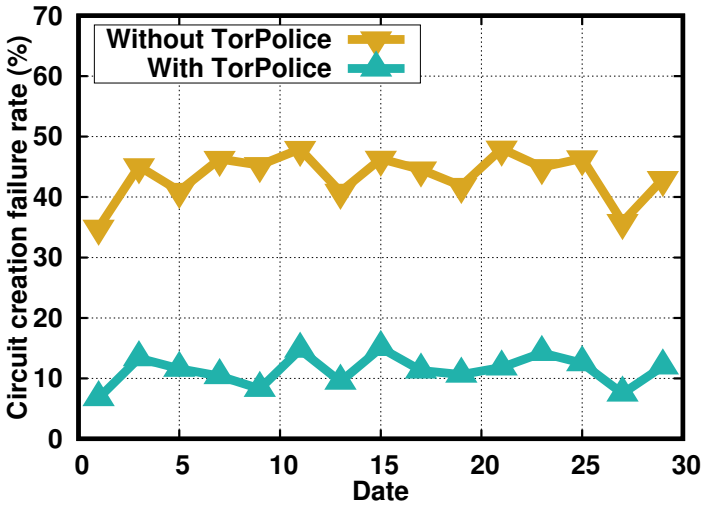


Figure 3.6: Circuit creation failure rates when Tor faced a multi-million node botnet C&C abuse. TorPolice can reduce the average failure rate by ~74%.

creation requests, an adversary can easily exhaust computation resources of the entire set of relays, driving in circuit failure ratios much higher than this threshold. Figure 3.7 demonstrates this vulnerability: a moderate-sized botnet with hundreds of thousands of bots is enough to cause very high circuit failure ratios. When Tor is protected by TorPolice, however, even a multi-million node botnet can only cause very limited failure rates for the current Tor network (represented by the consensus published on May 1, 2017).

3.10 Related Work

In this section, we discuss closely related work.

Capabilities in the Internet. Capability schemes ([11–13, 16, 89]) have been proposed to protect the Internet from DDoS attacks. In these approaches, capabilities specify certain traffic policing rules and meanwhile carry cryptographic signatures to ensure correctness. Victims (*e.g.,* servers or congested routers) police traffic based on received capabilities to stop attacks. Different from TorPolice, Internet capability designs do not consider privacy. Further, some of these capability schemes are difficult to deploy since they require modification of Internet core and client network stack. On the contrary, TorPolice is readily deployable in Tor with small overhead.

Anonymous Blacklisting Systems. Anonymous blacklisting systems [96] al-

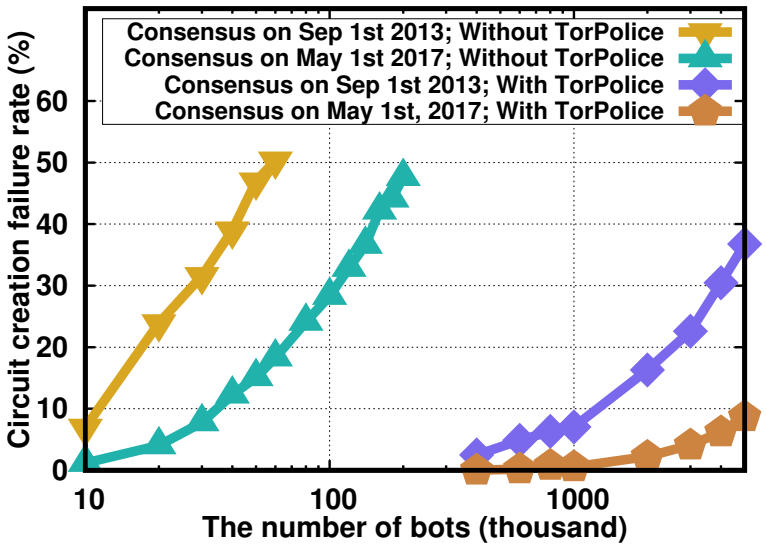


Figure 3.7: Without TorPolice, a moderate-sized adversary can paralyze Tor via cell flooding attacks. TorPolice can effectively mitigate this vulnerability.

low service providers to maintain a "blacklist" to explicitly block abusive users while serving non-abusive users without breaking anonymity. Anonymous blacklisting systems can be categorized into three broad groups: the pseudonym systems [77, 78, 97–99], the Nymble-like systems [100–104], and the revocable anonymous credential systems based on zero-knowledge proofs [105–107]. These systems either offer pseudonymity instead of full anonymity or require a trusted or semi-trusted authority to provide anonymity. TorPolice is not designed to be a new anonymous blacklisting system. Rather, TorPolice is explicitly designed for Tor, focusing on proposing a capability-based access control framework that allows service providers and Tor to enforce access rules to throttle various botnet abuses while still serving legitimate Tor users properly. Further, TorPolice's trust is more distributed since its AAs are fully distributed and each of them only has a partial view of the entire system.

Relay Incentives. Tor relay incentive mechanisms [108–111] are proposed to recruit more relays for the Tor network. Gold Star [108], BRAIDS [109] and LIRA [110] incentivize Tor clients to relay anonymous traffic by offering them prioritized Tor services. TorPath [111] instead pays relays Bitcoins. By allowing relays to redeem their received relay-specific capabilities for various benefits, TorPolice provides a general framework to support these incentive mechanisms. For instance, to support the similar incentive mechanism in [109], a relay $\mathbb{R}$ can redeem its received generic capabilities to obtain "prioritized relay-

specific capabilities" from the AAs. Then $\mathbb{R}$, as a client, can subsequently spend these prioritized capabilities to create premium Tor circuits to get premium services.

3.11 Chapter Summary

This chapter presents TorPolice, the first privacy-preserving access control framework that allows service providers and Tor to enforce self-selectable access policies on anonymous Tor connections so as to throttle various botnet abuses while still providing service to legitimate Tor users. TorPolice leverages blindly signed network capabilities to preserve the privacy of Tor users. We implement a prototype of TorPolice, and perform extensive evaluations to validate TorPolice's design goals.

CHAPTER 4

ENABLING WORK-CONSERVING BANDWIDTH GUARANTEES FOR MULTI-TENANT DATACENTERS VIA DYNAMIC TENANT-QUEUE BINDING

4.1 Introduction

Sharing the network of multi-tenant datacenters has been a critical theme for public clouds. The two primary objectives, among others, are bandwidth guarantees and work conservation. Bandwidth guarantees ensure predictable lower bound network performance for tenant applications. Recent studies show that, without bandwidth guarantees, network performance can experience 5x or more variations, leading to poor application performance [112]. Work conservation enables a tenant to use spare bandwidth beyond its minimum guarantee to further improve its application performance as well as boost provider network utilization. Given that datacenter traffic is bursty in nature and that the average network utilization is low [113–115], work conservation can deliver over 10x additional bandwidth to a tenant VM upon its minimum guarantee [116].

However, it is hard to achieve both bandwidth guarantees and work conservation simultaneously. Prior works such as Oktopus [112] and SecondNet [117] can achieve bandwidth guarantees, but they are not work-conserving. Seawall [118] and NetShare [119] achieve work conservation, but they do not provide bandwidth guarantees (more details in Section 2.9).

ElasticSwitch [116] takes the first step toward achieving both properties at the same time. It is an endhost based solution that first needs to translate per-VM hose-model bandwidth guarantees into VM-to-VM pair rate limiters (referred as Guarantee Partitioning, GP), and then dynamically allocates spare bandwidth to these VM pairs to achieve high utilization (referred as Rate Allocation, RA). However, this approach confronts two challenges. First, as tenant applications are typically agnostic to network operators, it is difficult for GP to accurately capture the real communication patterns among VMs. Second, to detect spare bandwidth, RA needs to probe the network by increasing rates,

which causes a tradeoff between accurately providing bandwidth guarantees and being work conserving [116, 120]: a conservative RA sacrifices work conservation, while an aggressive RA affects other tenants' bandwidth guarantees (see experiments in Appendix B).

Trinity [120] moves one step further to complement ElasticSwitch with simple in-network support. It exploits two priority queues in switches to segregate and prioritize the bandwidth guarantee traffic over work conservation traffic, so that aggressive RA of one tenant does not affect bandwidth guarantees of others. Trinity addresses the second challenge of ElasticSwitch as it eliminates the tradeoff, but it still inherits the challenge of GP since it still relies on endhost rate limiters to perform bandwidth guarantees and work conservation. Furthermore, it incurs additional issues such as packet reordering and starvation due to traffic segregation and priority queuing.

This thesis introduces QShare, a comprehensive in-network solution to address the above challenges. Instead of using two priority queues to segregate traffic for two different types, QShare directly leverages multiple weighted fair queues (WFQs) to slice network bandwidth for tenants. This ensures that (i) bandwidth guarantees are achieved through proper queue weight configuration and tenant placement rather than endhost rate limiters, thus relieving us of GP; (ii) the network link is driven to full utilization instantly as long as one tenant has sufficient demand; (iii) no matter how aggressively a tenant transmits, bandwidth guarantees of other tenants are not affected as they are served in separate weighted queues; and (iv) no packet reordering or starvation arises. While promising, QShare faces a practical challenge of queue scarcity—the number of queues on a commodity switch port (typically 8) can be less than the number of tenants served by this port (see Section 4.7.3.1 for details).

To address this challenge, we make the following observation: although the total number of embedded tenants associating with a port may be large, during a short time interval (*e.g.,* a few seconds), the number of concurrent tenants whose traffic demands exceed their bandwidth guarantees is small. This is also reflected by the measurement results in production datacenters, where the average link utilization is low [113–115]. Thus, to support more tenants with limited queues, QShare dynamically assigns dedicated queues for tenants with higher demands than their guarantees, while serving low-demanded tenants in a shared queue altogether.

QShare mainly contains two modules: a balanced tenant placement module

and a dynamic tenant-queue binding module (Section 4.3). The tenant placement module is responsible for allocating network resources to tenants to provide bandwidth guarantees. To facilitate the dynamic queue allocation for embedded tenants, our placement also tries to balance the usage of switch ports among tenants to avoid overwhelming certain ports. The tenant-queue binding module then takes into account the traffic demands of tenants and their payment factors to dynamically distribute queue resources among tenants.

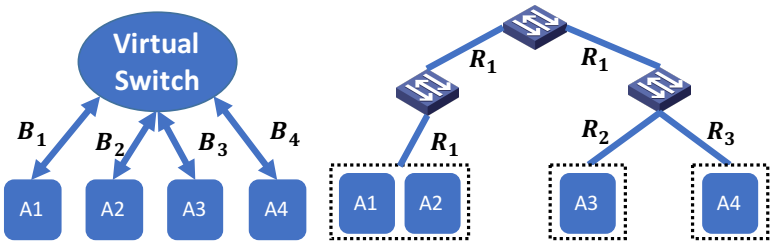
We implement a prototype of QShare with ~2000 lines of code (`C` for Linux kernel space and `Python` for user space), and perform extensive evaluations on testbed and via simulations. Our results suggest that:

- Without sacrificing bandwidth guarantees, QShare achieves (i) *perfect* work conservation given correct prediction on demand trends (not the exact demand), and (ii) over 91% link utilization given completely *unpredictable* demands.
- Given the above desirable properties, QShare significantly benefits applications, for instance, by reducing their flow completion times (FCTs) by up to 50% compared with the state-of-the-art [116, 120].
- With production datacenter settings, QShare can assign dedicated queues to ~90% of all embedded tenants even when the datacenter is fully reserved, yielding at least 3x throughput gain over the guarantees and better efficiency in link utilization.

4.2 Background And Motivation

4.2.1 Background

In multi-tenant datacenters [112, 117, 121, 122], a conceptually centralized *tenant manager* with global view of the datacenter state is responsible for managing all tenants, including tenant embedding, routing updates, logging, failure handling and recovery and so forth. By designing various components for the tenant manager, datacenter operators are able to achieve self-interested goals, such as accommodating more tenants and achieving efficient resource utilization. QShare can be viewed as a newly designed component in the tenant



(a) Hose model for B.G. (b) Bandwidth reservation

Figure 4.1: Figure 4.1(a) shows the tenant (VM) bandwidth guarantees defined in a symmetric hose model. Figure 4.1(b) shows the bandwidth reservation on each link after embedding the tenant: $\boldsymbol{R_1 = \min\{B_1 + B_2, B_3 + B_4\}}$; $\boldsymbol{R_2 = \min\{B_3, B_1 + B_2 + B_4\}}$; $\boldsymbol{R_3 = \min\{B_4, B_1 + B_2 + B_3\}}$.

manager to simultaneously accomplish the following two desirable properties: bandwidth guarantees and work conservation.

Network bandwidth guarantees are preferable properties in cloud computing to offer tenants predicable performance. A typical way to model bandwidth guarantees is the hose model [112, 116, 121, 123–126]. As an illustrative example, Figure 4.1(a) shows a tenant `A`'s bandwidth guarantees defined in a hose model, and Figure 4.1(b) illustrates the reserved bandwidth on each physical link to satisfy the bandwidth guarantees after tenant embedding. For simplicity, a symmetric hose model is plotted in Figure 4.1. Providing accurate bandwidth guarantees for VMs that can use multiple paths is an open problem since it requires a *perfect* load balancer to accurately distribute each VM's traffic over multiple paths such that the sum of guarantee on each path equals to the total amount of guaranteed bandwidth. As a result, prior proposals for providing bandwidth guarantees are either within the scope of tree-based network topology [112, 116, 117] or confining each tenant's traffic within a tree in multi-path network topologies [5, 121]. QShare belongs to the second category as typical datacenters (*e.g.,* fattree [127]) are built with path redundancy. QShare, however, can still fully utilizes the redundant network links via balanced tenant placement.

Work conservation is desired for achieving efficient resource utilization. Formally, in the context of multi-tenant datacenters, work conservation is defined as follows: for any link L in the network, as long as there exists at least one tenant that has packets to send along link L, L cannot have spare bandwidth [124]. We note that work-conservation does not guarantee that there are no idle links in the network. Idle links may exist due to the lack of traffic demands or high-

demanded tenants are bottlenecked by other links.

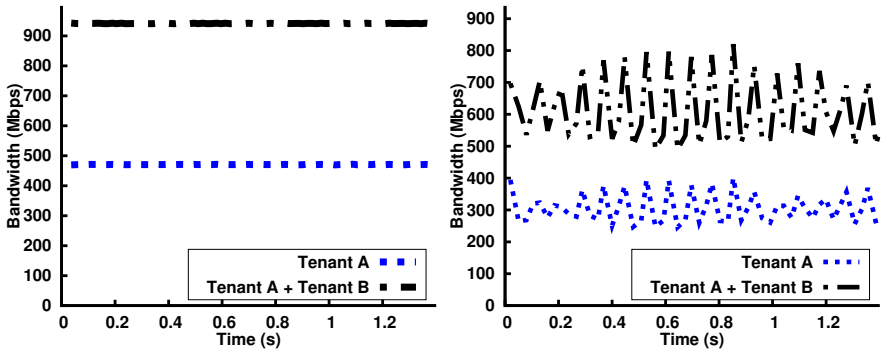
4.2.2 State-of-the-Art Solutions

ElasticSwitch [116] makes the first attempt to achieve work-conserving bandwidth guarantees. It is an end-host based solution composed of two modules: a Guarantee Partitioning (GP) module that divides VM X's hose-model guarantee into guarantees to/from each other VM that X communicates with, and a Rate Allocation (RA) module that assigns spare bandwidth to these VM pairs to achieve high network utilization. However, it suffers from the following two key challenges.

First, since the traffic matrix (TM) among the VMs of a tenant is typically agnostic, GP has to gradually learn each VM-pair's demand via periodic source-destination VM coordination and throughput measurement. Whenever the TM changes, GP needs to re-estimate the TM even if per-VM demand remains the same (see illustrative example in Section B). Given highly bursty and dynamic TM in datacenters, it is challenging for the GP to capture the real communication pattern and estimate the TM correctly, especially considering that tens of thousands of VMs can produce billions of VM pairs. Further, at such scale, the overhead of maintaining these VM-pair rate limiters at hypervisors is non-negligible [116].

Second, RA in ElasticSwitch [116] aims to grab available network bandwidth beyond the provided guarantees. It probes the network by increasing rates, detects congestion via packets losses or ECN, and then allocates the spare bandwidth to VM pairs in max-min fashion following weighted TCP algorithms [118, 128]. As mentioned in [116, 120], it has a tradeoff between accurately providing bandwidth guarantees and being work-conserving: aggressive RA could affect other tenants' guarantees whereas conservative RA ends up with bandwidth waste. In practice, RA's performance depends on the parameter choice and system tuning.

Trinity [120] moves one step further to complement the endhost based ElasticSwitch with simple in-network support. It exploits two priority queues in switches to segregate and prioritize the bandwidth guarantee traffic over work conservation traffic. As a result, VMs can send work-conservation traffic more aggressively without affecting bandwidth guarantees of others. Thus, Trinity



(a) In-network Support (b) Pure endhost-based

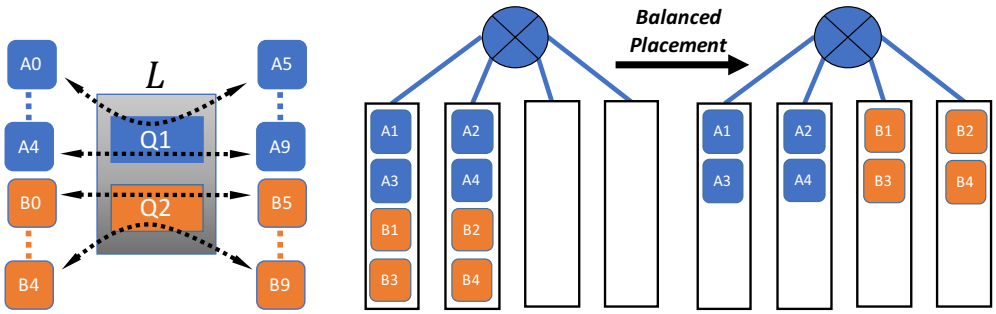
Figure 4.2: Compared with pure endhost-based solution, QShare achieves perfect (no tradeoff) work-conserving bandwidth guarantees via in-network support.

achieves work conservation in a static context, *i.e.*, the demand of each VM-pair is a priori knowledge. However, it inherits the challenges of GP in dynamic context since it still needs to translate per-VM bandwidth guarantees into VM-pair rate limiters on hypervisors. Further, since network traffic is segregated and served with strict priorities, Trinity raises packet reordering and starvation issues in practice.

4.3 QShare Overview

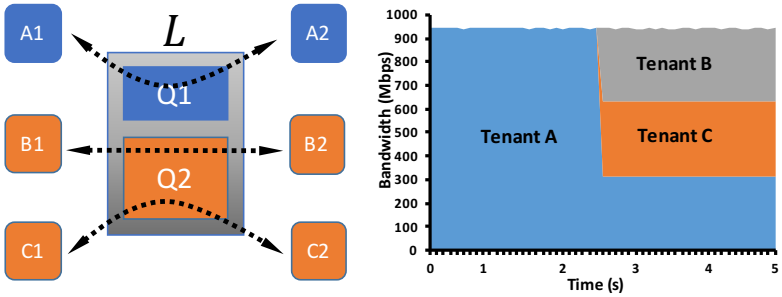
QShare is a comprehensive in-network solution to address the above challenges. Instead of using two priority queues to segregate traffic for two different types, QShare directly leverages multiple weighted fair queues (WFQs)[1] to slice network bandwidth for tenants. This enables QShare to provide tenant-level bandwidth guarantees and work conservation (instead of rigid VM-to-VM pair level as in both ElasticSwitch [116] and Trinity [120]), thus leaving tenant applications full flexibility to use the allocated bandwidth as needed. We note that such tenant-level bandwidth guarantees are also used in [112, 129], but they fail to achieve work conservation.

[1]The WFQ is emulated by WRR on some types of switches.



(a) In-network support

(b) Balanced tenant placement



(c) Tenant-queue allocation

Figure 4.3: Three illustrative examples for QShare's design. Figure 4.3(a) shows that QShare incorporates the in-network WFQ support. Thus, tenants A and B sharing the link L are served in two separate weighted queues. Figure 4.3(b) shows that QShare's tenant placement algorithm balances the usage of switch ports among the embedded tenants. Figure 4.3(c) shows that assigning a dedicated queue for tenant A with sufficient traffic demand achieves perfect work conservation. Meanwhile, tenant B and C's bandwidth guarantees are immediately satisfied once they become active.

4.3.1 In-Network Support

WFQ on commodity switches offers desirable in-network support for achieving work-conserving bandwidth guarantees. We use the following experiment to demonstrate its benefit. We place two tenants A and B, both provisioned with 10 VMs, on our testbed. Each tenant's VMs are evenly distributed across two racks connected by a core link with one Gbps capacity. As A and B share the core link L, their flows are served in two separate queues whose weights are configured proportionally to their guaranteed bandwidth on the link (Figure 4.3(a)).

Consider a case where both A and B adopt the same symmetric hose model, in which each VM is guaranteed 50 Mbps bandwidth. Thus, both tenants have 250 Mbps guaranteed bandwidth on the core link. To generate traffic, each VM

in one rack is configured to communicate with randomly selected VMs, using our client/server program described in Section 4.7. Each VM's demand and its communication pattern are completely random. Only intra-tenant communication is considered. We measure the amount of core-link bandwidth utilized by each tenant. As plotted in Figure 4.2(a), without relying on any TM estimation, QShare achieves perfect work-conserving bandwidth guarantees without imposing packet reordering or starvation issues. We repeat the experiment using self-implemented prototype of ElasticSwitch. As illustrated in Figure 4.2(b), we notice a significant gap (over 300 Mbps) between the aggregate bandwidth of A and B and the link capacity, *i.e.,* over 60% of the unreserved bandwidth is wasted. We are aware that ElasticSwitch's performance depends on parameter settings. We consider different settings in Appendix B.

4.3.2 Design Overview

The key challenge of QShare is to address the problem of queue scarcity: the number of queues on each switch port (typically 8) can be less than the number of tenants served by this port so that we cannot allocate a dedicated queue for each tenant. To address this challenge, QShare designs two modules: a balanced tenant placement module and a dynamic tenant-queue binding module.

The placement module first seeks to provision tenant network to ensure bandwidth guarantees. Further, it *balances* the usage of switch ports among tenants to reduce the stress of performing the dynamic queue allocation in the binding module. For instance, if both placements in Figure 4.3(b) satisfy bandwidth guarantees, QShare prefers the one on the right side since the switch ports (and their queues) are more evenly utilized by the tenants.

The tenant-queue binding module dynamically assigns dedicated queues to tenants whose demands are higher than their guaranteed bandwidth, and meanwhile serves all the low-demanded tenants in a shared queue (they may employ ElasticSwitch-like rate allocation to improve the worst case performance, as explained below). As a result, tenants in dedicated queues can burst their traffic in arbitrary communicate patterns without affecting other tenants. This design is the key to avoid the challenging GP and to eliminate the tradeoff between bandwidth guarantees and work conservation in the endhost based solutions [116, 120]. We perform an experiment to show this. Consider that

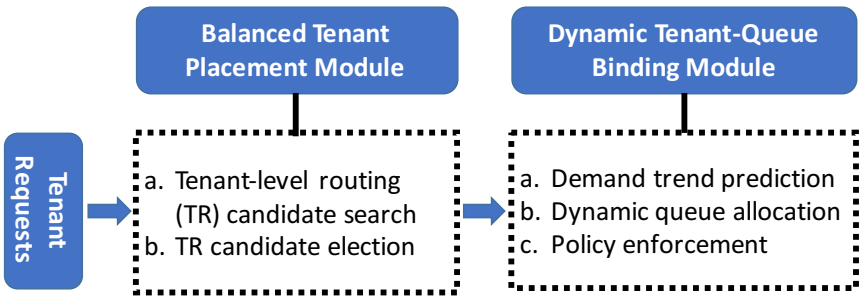


Figure 4.4: The architecture of QShare.

three tenants A, B and C compete on a link L with one Gbps capacity, shown in Figure 4.3(c). Each tenant has 300 Mbps guarantee on L. Assume L can support two queues. Suppose tenant A is high-demanded so that QShare assigns it a dedicated queue (with weight one) whereas B and C share a common queue (with weight two). When only A is active, it fully utilizes the link capacity, achieving work conservation. Further, B and C are not overwhelmed by A, and immediately receive their guaranteed bandwidth once becoming active.

One challenge of the binding mechanism is how to assign right tenants dedicated queues since traffic demand is dynamic. QShare addresses the challenge as follows. First, rather than predicting traffic matrix for each tenant as proposed in [116, 120], QShare's demand prediction relies on only a scalar metric (detailed in Section 4.5.1) of each tenant, which greatly reduces stress of prediction. Second, to improve the worst case performance when traffic demand prediction is inaccurate and high-demanded tenants are mistakenly placed in the shared queue, QShare can employ ElasticSwitch for tenants in the shared queue to achieve moderate work-conserving bandwidth guarantees in the spirit of ElasticSwitch. Finally, we perform testbed experiments (Section 4.7.1) to quantify effects of the binding mechanism: (i) the average utilization deficit caused by binding errors is less than 9% of the total capacity; (ii) to achieve good performance, it is sufficient to perform dynamic binding at more coarse time granularity (*e.g.,* a few seconds) compared with the traffic matrix estimation performed at the granularity of milliseconds in [116, 120].

We plot the system architecture in Figure 4.4. Next we briefly discuss the components of each module, and defer their design details in Section 4.4 and Section 4.5, respectively.

4.3.2.1 Balanced Tenant Placement Module

The balanced tenant placement module has two components. In particular, given a tenant request, the routing explorer (Section 4.4.1) outputs all Tenant Routing (TR) candidates that can accommodate the tenant. The placement algorithm (Section 4.4.2) evaluates each candidate to select the most desired one.

Tenant Routing (TR) Exploration. As explained in Section 4.2.1, a tenant **T**'s TR is a tree in the physical network topology that connects the servers/hypervisors hosting **T**'s VMs ("servers" and "hypervisors" are used interchangeably). Traffic generated by **T**'s VMs is confined within its TR. Thus, the TR needs to be provisioned with sufficient VM slots and network bandwidth to fulfill **T**'s requirement. TR exploration, essentially, is the topology search process that produces a set of virtual networks (*i.e.,* overlay trees) that can accommodate **T**. Admission rules are applied here to accept new tenants only if the datacenter has sufficient spare capacities.

TR Candidate Election. Each TR candidate is evaluated based on two criteria: bandwidth reservation cost and queue occupation cost. Reducing bandwidth reservation cost allows datacenters to accommodate more tenants, while the key reason for considering the queue occupation cost is to reduce the management stress for the dynamic tenant-queue binding module.

4.3.2.2 Dynamic Tenant-Queue Binding Module

The dynamic binding module executes *periodically* to distribute queues among tenants. It is built on (i) tenant demand trend prediction (Section 4.5.1), (ii) the queue-to-tenant allocation algorithm (Section 4.5.3) and (iii) the policy enforcer enforcing allocation decisions inside the network (Section 4.5.4).

Traffic Demand Trend Prediction. Based on the usage measurement in current *control interval*, QShare predicts demand trends of tenants in the next interval, *i.e.,* whether a tenant tends to have higher demand than its guarantees in the next interval. QShare's prediction relies on only a scalar metric rather than the per-VM pair traffic matrix as proposed in [116, 120].

Queue-to-Tenant Allocation. The allocation algorithm dynamically distributes queues among tenants. In case of queue scarcity, it ranks the competing tenants based on both their demands and payment. Considering payment mitigates the problem of real demand lying by tenants.

Policy Enforcer. The tasks of policy enforcer include performing network-related operations (*e.g.,* switch configuration), tagging tenant packets with proper dscp values and running ElasticSwitch-like rate allocations at hypervisors for tenants without dedicated queues.

4.4 Balanced Tenant Placement

The goals of tenant placement are (i) provisioning virtual networks for tenants to satisfy their computation and bandwidth guarantees and (ii) balancing the overall switch queue utilization among tenants. The placement algorithms proposed in [112, 121] aim to maximize the number of accepted tenant requests, which is an NP-hard problem similar to [123]. Different from their algorithms that make local embedding decisions (*i.e.,* embed a tenant immediately once a feasible option is found), our balanced tenant placement requires *global topology investigation*, *i.e.,* evaluating all feasible options before making embedding decisions. Thus, our placement algorithm, formulated in Algorithm 2, includes two parts (i) TR candidate exploration and (ii) candidate election.

4.4.1 TR Candidate Exploration

We first explain TR candidate exploration in the widely adopted multi-rooted tree datacenter topology [5, 121, 127, 130, 131]. Then, we discuss extending such exploration to support randomly connected topology [132, 133].

Given a tenant request, Algorithm 2 explores the topology from the lowest layer (hypervisor layer) toward the highest layer (core switch layer). At each layer, function get_TRs_at_layer (line 7) obtains all TR options at this layer. A layer-i TR option is a tree rooted at layer i. Its leaves are the servers reachable from the root using only downward paths. Then the algorithm evaluates these TRs to produce *feasible* ones, called TR candidates (line 9). Generally speaking, a TR option is feasible if it has enough capacity to accommodate the tenant. Function evaluate_TR, detailed in Section 4.4.2, determines such feasibility. If no TR candidates can be found, the algorithm continues exploration in the next layer (line 11). Otherwise, it stops further exploration and returns the desired TR elected from all candidates (line 15) using the criteria described in Section 4.4.2. The early return confines tenants at the lowest possible layer to

Algorithm 2: Balanced Tenant Placement

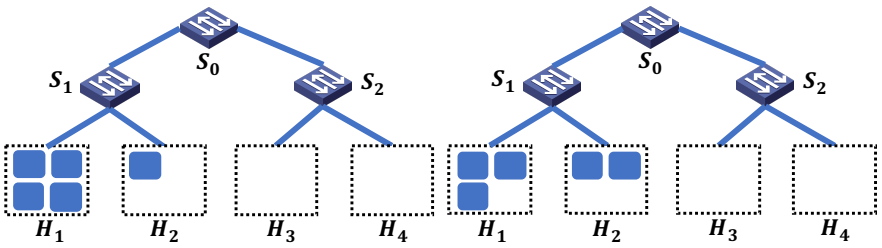
```
1  Input: A tenant request with explicit guarantees.
2  Output: The desired TR or an embedding error.
3  Main Procedure:
4  begin
5      layer ← 1;
6      while True do
7          TRs ← get_TRs_at_layer(layer);
8          for T ∈ TRs do
9              [feasible, cost] ← evaluate_TR(T);
10             if feasible then TR_candidates.add((T, cost));
11         if TR_candidates is empty then
12             layer ← layer + 1;
13             if layer > n then return False;
14         else
15             return get_desired_TR(TR_candidates)
16 Function: evaluate_TR(T):
17     OA ← get_optimal_allocation(T);
18     if OA is feasible then return [True, (c_b, c_q)];
19     else return [False, null];
```

avoid unnecessary network usage at higher layers. If no TR candidates can be found after exploring the entire topology with n layers, the algorithm returns false (line 13), indicating an embedding error due to the lack of resources.

Random Topology. To support random topology, Algorithm 2 adopts the k-shortest path algorithm [134] to obtain a set of paths between each hypervisor pair and then combines them to produce TR options. k, similar to $layer$ in Algorithm 2, determines TR exploration space.

4.4.2 TR Evaluation and Candidate Election

A TR option is feasible if (i) the total available VM slots from all its servers are enough to hold the tenant's VMs and (ii) each link of the TR has enough available capacity to satisfy the tenant's bandwidth guarantees. Although evaluating the first rule is straightforward, the second rule requires more investigation. In particular, given a TR option, the amounts of bandwidth required on its links depend on the VM locations inside the TR. Specifically, consider a homogeneous hose model where all VMs have the same inbound and outbound bandwidth guarantee **B**. Given a link L of the TR, removing L breaks the TR into two



(a) Requring **B** on $S_1 \leftrightarrow H_1$ (b) Requring 2**B** on $S_1 \leftrightarrow H_1$

Figure 4.5: Given the TR rooted at S_1, the bandwidth required on its link depends on VM locations inside the TR.

disjoint components. If m VMs are in one component and n VMs are in the other one, then the bandwidth required on L is $\mathbf{B} \cdot \min\{m, n\}$. Figure 4.5 plots a TR rooted at S_1. For the VM location in Figure 4.5(a), the two links ($S_1 \leftrightarrow H_1$ and $S_1 \leftrightarrow H_2$) both need to reserve **B** ($\min\{\mathbf{B}, 4 \cdot \mathbf{B}\}$) whereas they have to reserve 2**B** for the VM location in Figure 4.5(b).

To reduce the total network bandwidth required for embedding the tenant, function get_optimal_allocation (line 17) produces the VM location that requires the least bandwidth reservation. For homogeneous hose models, the optimal allocation is produced as follows: (i) find the server H in the TR with the largest *usable VM slots*, (ii) allocate as many VMs as possible to H, (iii) update the remaining network/server capacity after allocation and (iv) repeat step one until either all VMs are allocated (indicating feasibility) or all servers in the TR have been investigated (indicating infeasibility). The usable VM slots for H in the TR is restricted by both the available VM slots in H and the available bandwidth on the path from H to the TR's root. For instance, in Figure 4.5, if we assume the available bandwidth on link $S_1 \leftrightarrow H_1$ is less than **B**, the usable VM slots in H_1 is 0, rather than four.

If the TR's optimal allocation is feasible, it becomes a candidate for embedding the tenant. Algorithm 2 then computes its bandwidth cost c_b as the sum of reserved bandwidth for the tenant on each link of the TR, and the queue occupation cost c_q as the largest number of tenants served by any of the TR's links (line 18).

Candidate election is based on both c_b and c_q. Each TR candidate is associated with a $cost$ combining c_b and c_q. The desired TR is the one with lowest $cost$. One strategy for computing $cost$ is assigning more weight to c_q when the datacenter load is light to prefer more balanced placement whereas assigning

more weight to c_b for heavy-loaded network to prefer the placement with fewer bandwidth cost.

Supporting High Availability. Algorithm 2 can be extended to support high availability [135]. In [135], the worst-case survival ratio (WCS) is defined as the smallest fraction of VMs remaining functional during a single point failure. Consider server as the fault domain. Given a tenant with N VMs and WCS as f, one server can host at most $(1-f)N$ VMs for this tenant. By patching the constraint in function get_optimal_allocation, Algorithm 2 can produce TR that satisfies the high availability requirement.

Search Complexity. The search complexity for embedding tenants depends on the layers at which Algorithm 2 returns. In a fattree topology [127], the worse case complexity (*i.e.,* the algorithm returns at the core switch layer) is $O(V^{\frac{5}{3}})$, where V is the number of nodes in the network. For topologies built with higher over-subscription ratios than fattree, the search complexity is smaller as the number of TR options at each layer is smaller. Further, the topology search results can be cached to achieve long-term efficiency [136].

4.5 Tenant-Queue Binding

To support more tenants with limited number of queues, QShare's design is inspired by how the working set of a process is often much smaller than the total memory it consumes. Similarly, only tenants whose traffic demands exceed their bandwidth guarantees need dedicated queues. Thus, there is an opportunity for QShare to dynamically allocate limited number of queues to high-demanded tenants. In particular, QShare periodically evaluates each tenant and allocates queues among tenants based on their *scores*. Each tenant's score encapsulates its *usage factor* (Section 4.5.1) and *payment factor* (Section 4.5.2) so as to prioritize high-demanded and honest tenants.

4.5.1 Tenant Demand Trend Prediction

As prior works [116, 120] rely on Guarantee Partitioning (GP) to achieve bandwidth guarantees, they need to predict each tenant's traffic matrix, *i.e.,* per VM-pair traffic demand. However, since tenant applications are often agnostic to network operators, it is challenging to capture the real communication patterns

among VMs and predict traffic demand between each VM pair. On the contrary, QShare's tenant-queue binding module only needs to predict whether a tenant tends to have higher demands than its guaranteed bandwidth. Thus, rather than predicting traffic matrix, QShare proposes to use a scalar metric, usage factor (U-factor), to indicate a tenant's network utilization with respect to its guaranteed bandwidth. We do not claim that U-factor is the optimal metric for demand prediction. However, it does greatly reduce the stress of prediction by focusing on tenant-level *demand trend* rather than VM-level traffic matrix.

Each tenant's U-factor is computed per *control interval*. Specifically, in each control interval, all hypervisors measure the bandwidth utilization of their hosted VMs. As VMs can have both inbound and outbound traffic, bi-directional bandwidth usage is considered. For instance, consider a hypervisor H_j hosting m VMs of a tenant **T**. Then **T**'s inbound (outbound) bandwidth usage U_j^i (U_j^o) measured by H_j is the sum of inbound (outbound) bandwidth usage from all these m VMs.

At the end of each control interval, QShare computes each tenant's U-factor. For tenant **T**, one way of computing its U-factor $\mathbf{U_T}$ is as follows

$$\mathbf{U_T} = \min\{\max_{H_j \in \mathbf{H}} \frac{\max\{U_j^i, U_j^o\}}{B_j}, 1\}, \tag{4.1}$$

where **H** is the set of hypervisors hosting **T**'s VMs and B_j is **T**'s guaranteed bandwidth on H_j's network interface. If H_j hosts m VMs from **T** (provisioned with total N VMs), $B_j = \mathbf{B} \cdot \min\{m, N-m\}$ considering a symmetric and homogeneous model with per-VM guarantee **B**.

The design rationale of Equation (4.1) is as follows. The innermost max is necessary as the high-demanded VMs may either send or receive large volumes of traffic. The middle max is designed to handle many-to-one traffic pattern in which many source VMs in remote servers are communicating with a few destination VMs in the local server. Although source hypervisors may measure small usage since source VMs are bottlenecked by destination VMs. **T** actually has large traffic demand at these receivers. Taking the largest usage among all hypervisors will capture such a communication pattern. Finally, the outermost min sets a $\mathbf{U_T}$ cap of one. We leave the exploration of other U-factor definitions in future work.

4.5.2 Tenant Lying Mitigation

Merely using U-factors to allocate queues has problems when tenants lie about their real bandwidth guarantees: a tenant can deliberately request smaller guaranteed bandwidth so as to have high U-factors. Note that no work-conserving allocation policies can completely prevent tenants from gaining advantages via lying, *i.e.,* being strategy-proof [124]. To mitigate the problem caused by lying, QShare proposes to consider payment factors, along with U-factors, when scoring tenants. Each tenant's payment factor and its guaranteed bandwidth are positively correlated such that deliberately requesting lower guarantees reduces a tenant's score whereas exaggerating guarantees requires higher payment. As designing pricing model is not the focus of this chapter, QShare assumes, for simplicity, that a tenant's payment factor is proportional to the total guaranteed bandwidth required by its hose model.[2] Thus, given tenant $\mathbf{T}$ with N VMs and each VM requests guaranteed bandwidth $\mathbf{B}$, its payment factor is $kN\mathbf{B}$, where k is a constant depending on the pricing model.

Simplifying $\mathbf{U_T}$ in Equation (4.1) as $\min\{\frac{U_m}{B_m}, 1\}$, then tenant $\mathbf{T}$'s score $S_\mathbf{T}$ is computed as follows

$$S_\mathbf{T} = kN\mathbf{B} \cdot \mathbf{U_T} = \begin{cases} \tilde{k}U_m, & \text{if } \mathbf{U_T} < 1 \\ kN\mathbf{B}, & \text{otherwise.} \end{cases} \tag{4.2}$$

$\tilde{k} = kN\mathbf{B}/B_m$, where B_m is determined by the number of $\mathbf{T}$'s VMs hosted by hypervisor H_m.

Using $S_\mathbf{T}$ as the criterion for queue allocation can mitigate problems caused by lying. On the one hand, as $S_\mathbf{T}$ is bounded by $kN\mathbf{B}$, deliberately requesting smaller $\mathbf{B}$ would result in a lower cap of $S_\mathbf{T}$, which is disadvantageous when competing with other tenants. On the other hand, deliberately requesting higher $\mathbf{B}$ than real demand also has problems as (i) tenant $\mathbf{T}$ would have to pay more and (ii) its $S_\mathbf{T}$ would be determined by $\mathbf{T}$'s real usage rather than its claimed guarantees if $\mathbf{T}$ has smaller demands than its guarantees (*i.e.,* $\mathbf{U_T} < 1$). Generally, high-demanded tenants are preferred since $S_\mathbf{T}$ is non-decreasing as bandwidth usage increases, which is desirable for queue allocation.

[2] Payment for computation resources is not considered as QShare focuses on bandwidth management.

Algorithm 3: Queue Allocation Algorithm

```
1  Input: The set of embedded tenants 𝕊.
2  Output: Tenant-queue assignment.
---
3  Sort the tenants in 𝕊 decreasingly by their scores;
4  for T ∈ 𝕊 do
5      if T has a dedicated queue then continue;
6      else if T's TR has a spare queue then
7          enqueue_tenant(T);
8      else opportunistically_enqueue(T);
9      Update queue allocation state;
10 Queue weight computation;
---
11 Function: enqueue_tenant(T):
12 for L ∈ T's TR do
13     reserved_bandwidth ← B · min{m, N − m};
---
14 Function: opportunistically_enqueue(T):
15 for L ∈ T's TR do
16     get_opportunistic_queues_from_LSTs(L);
17 if T's TR has an opportunistic queue then
18     enqueue_tenant(T);
```

4.5.3 Dynamic Queue Allocation

We present the queue allocation logic in Algorithm 3. A tenant is assigned a dedicated queue only if it is assigned a dedicated queue on each link of its TR. Otherwise, the tenant will be served in the shared queue on each link of its TR. To prioritize tenants with higher scores, Algorithm 3 starts queue assignment from the tenant with the highest score, breaking tie randomly (line 3).

If a tenant T already occupies a dedicated queue, it continues to hold the queue (line 5) for the next control interval. This indicates that T maintains its high score or owns a dedicated queue on each link of its TR due to the lack of queue contention, which is possible due to the balanced placement (see analysis in Section 4.7.3.1).

If T is currently placed in the shared queue, Algorithm 3 determines whether allocating T a dedicated queue is possible. To satisfy the condition on line 6, each link of T's TR needs to have at least one spare queue. If positive, function enqueue_tenant assigns T a queue on each link L of its TR (line 12).

Finally, if at least one link of T's TR runs out of queues, function opportunistically_enqueue (line 8) *opportunistically* finds queues for T by preempting queues from low-scored tenants (LSTs). Specifically, on link L without spare

queues, the algorithm obtains an opportunistic queue occupied by a tenant $\tilde{T}$ such that (i) $\tilde{T}$'s score is less than T's score and (ii) $\tilde{T}$'s score is the smallest among all tenants owning a queue on L (line 16). If an available queue, either opportunistic or unoccupied, exists on each link of T's TR, we say that T's TR has an opportunistic queue (line 17), and then enqueue T. The dequeued tenants will be served in shared queues during the next control interval.

Queue allocation state is updated after handling T (line 9). Once queue allocations for all tenants are finished, QShare computes weight for each queue (line 10). For a queue $\mathcal{Q}_i$ on link L, its normalized weight is the ratio of reserved bandwidth in $\mathcal{Q}_i$ to the total reserved bandwidth on link L. In practice, weights need to be proportionally translated into the supported values (*e.g.,* one to 15 on our switches).

Tenant departures will trigger state update as well. Newly arrived tenants are served in shared queues, and will be evaluated at the end of current interval.

4.5.4 Policy Enforcer

To enforce queue allocation decisions inside the network, QShare needs to perform (i) packet tagging and (ii) network configuration. Packet tagging is to ensure that packets are served in correct queues. We use dscp tagging to achieve this. To avoid ambiguity, the D-tenants (tenants with dedicated queues) whose TRs share at least one common link cannot use the same dscp value. D-tenants whose TRs are non-overlapping can reuse the same dscp value. Given that dscp values range from zero to 63, finding the smallest possible number of dscp values in a legal assignment can be reduced to the k-coloring of a graph, which is NP-hard [137].

To address the dscp usage concerns, we analyze the efficiency of a greedy assignment in large scale datacenters based on production datacenter settings. The results show that 64 dscp values are sufficient to avoid conflict even when the datacenter is fully reserved (see details in Section 4.7.3.1). Further, technically, it is possible to mutate dscp values on switch ports via the dscp-to-dscp mutation map [138]. Thus, based on a dscp mapping on each port, a tenant can use different dscp values on different ports, eliminating the static dscp value reservation required on each link of its TR, which in turn eliminates the possibility of dscp conflict.

Network configuration involves configuring queues on each link with proper weights and `dscp` values, which requires WFQ configuration on both ports of the link. For edge links connecting servers and switches, software WFQ is required on hypervisors. To support automation, QShare designs a network action container to perform configuration in a batch: operations on different switches are parallelized via multi-threading so that the marginal configuration latency is negligible.

The final part of the policy enforcer is that QShare can run ElasticSwitch-like rate allocation mechanisms [118, 128] for tenants without dedicated queues to provide them bandwidth guarantees and achieve moderate work conservation in the worse case when all D-tenants have insufficient demands. However, QShare imposes smaller overhead than ElasticSwitch [116] since it only performs rate allocations for tenants without dedicated queues.

4.6 Implementation

The prototype of QShare contains both user-space and kernel-space programs, as shown in Figure 4.6. The user-space programs, executed globally, are responsible for managing the whole datacenter whereas the kernel-space program, running on each hypervisor, manages the local hypervisor. Two spaces interact with each other such that queue allocation decisions are made based on the distributed measurement reported by all hypervisors, and meanwhile the allocation decisions are pushed back to the kernel modules for enforcement on hypervisors. The implementation has ~2000 lines of code (Python in user space and C in kernel space).

The user-space programs include tenant placement, queue allocation and the network action container. The kernel-space module, built on NetFilter [139], includes tenant traffic monitor, rate allocation (for tenants in shared queues), software WFQ (for tenants with dedicated queues) and packet dscp tagging. On each hypervisor, a user-space deamon (not plotted) based on Netlink [140] interacts with the kernel module.

Note that implementing a hypervisor that can support all kinds of VM management is out of this chapter's scope. Our prototype builds a simple hypervisor that can support QShare-related operations, such as identifying the VMs of each tenant.

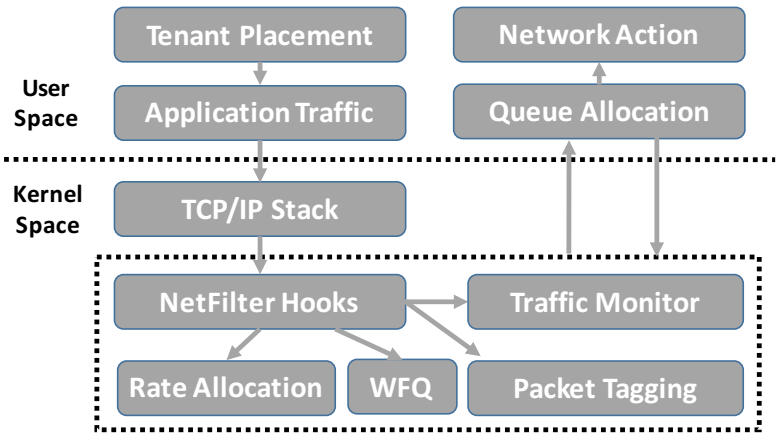


Figure 4.6: The software implementation of QShare.

4.7 Evaluation

Our evaluation centers around the following questions:

(i) How does traffic dynamic affect QShare's performance? With correct predictions on demand trend (not the exact demand), QShare achieves *perfect* work-conserving bandwidth guarantees: all bandwidth guarantees are satisfied and meanwhile the bottleneck link is fully utilized (Section 4.7.1.1). Even when demand trends are *completely unpredictable*, QShare drives the bottleneck link to over 91% utilization (Section 4.7.1.2) without comprising bandwidth guarantees.

(ii) How well can QShare benefit applications? Given the above desirable properties, QShare significantly benefits applications, for instance, by reducing their flow completion times (FCTs) by up to 50% compared with the state-of-the-art solutions [116, 120] (Section 4.7.2).

(iii) How well can QShare manage large-scale datacenters? Based on observations from production datacenters, we analyze QShare in a large scale datacenter. We show that QShare can assign dedicated queues to ~90% of the tenants in any control interval even when the datacenter is fully reserved. Thus, QShare produces at least 3x throughput gain over the guarantees and achieves higher efficiency in link utilization (Section 4.7.3).

(iv) How much overhead does QShare impose? QShare imposes small overhead for switch configuration, running rate allocations and embedding tenants (Section 4.7.4).

Testbed Experiment Setup. We build a physical testbed containing 10 servers

and each server provisions 10 VM slots, for a total of 100 VMs. Each server installs a Gigabit Ethernet NIC and runs the 3.13.0 Linux kernel. We evenly distribute the servers into two racks inter-connected by two Pronto-3297 48-port Gigabit (ToR) switches. Thus, the topology is 5:1 oversubscribed and the core link may be congested when VMs are sufficient demands. Each port supports up to eight WFQ queues. We embed multiple tenants in the testbed, with random sizes from two to 20 VMs.

We develop a client/server program to generate traffic. The clients initiate long-lived TCP connections to randomly selected servers and request flow transmission. All VMs run both the client and server programs. Only intra-tenant communication is allowed.

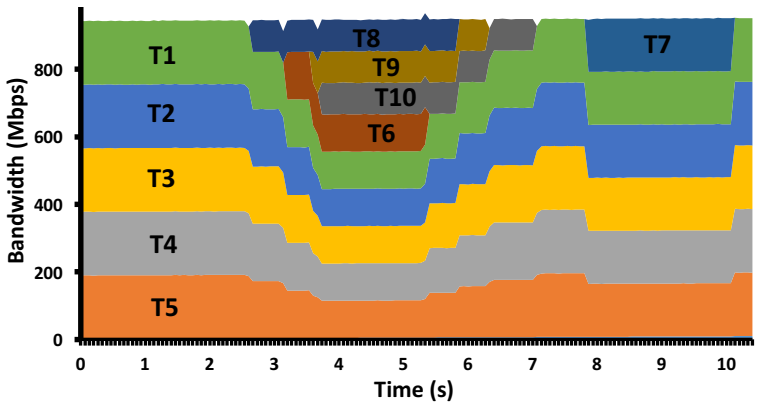
4.7.1 Work-Conserving Bandwidth Guarantees

In this section, we consider how traffic dynamics may affect QShare's performance for enabling work-conserving bandwidth guarantees. We consider the following two scenarios. The first case is that a tenant's demand trend is predictable: *i.e.,* once a tenant has high traffic demand, this trend continues for few seconds. Trend predictability is not over-optimistic since hot spots in production datacenters can last over tens of seconds [114]. The second case is that the demand trend is *completely unpredictable*: *i.e.,* a tenant's future demands are independent on its current or previous demands. In both cases, QShare does not impose any constraint on VM communication patterns, *i.e.,* one client can request flow transfers from arbitrary servers at any time.

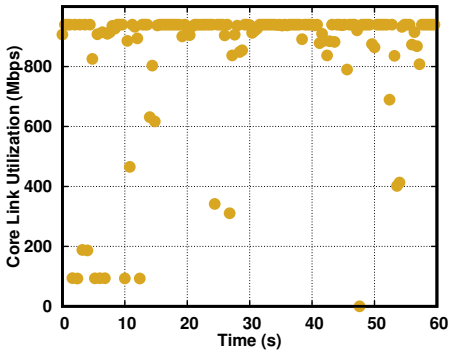
To quantify the worst-case performance degradation caused by traffic unpredictability, we first disable the ElasticSwitch-like rate allocations for the tenants without dedicated queues, and allocate them at most their guaranteed bandwidth.

4.7.1.1 Predictable Demand Trend

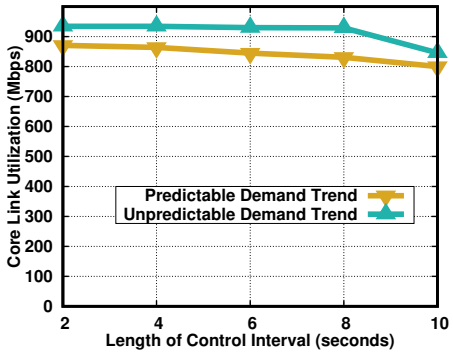
In this experiment, we consider 10 tenants competing on the core link. Each tenant is guaranteed 94 Mbps bandwidth on the core link. To generate traffic, we randomly pick five tenants (referred to as T1 to T5) as high-demanded tenants whose clients request sufficient flow transfers during our measurement



(a) Per-tenant runtime bandwidth utilization given predictable demand trends.



(b) Aggregate core link utilization given unpredictable traffic trend.



(c) Avg core link utilization with varying control intervals.

Figure 4.7: Figure 4.7(a) plots runtime bandwidth utilization of all tenants given correct demand prediction. QShare achieves perfect work-conserving bandwidth guarantees in this case. Figure 4.7(b) plots the total runtime utilization given completely unpredictable demands. Only few under-utilized cases are observed during the measurement period, yielding over 91% average utilization. Figure 4.7(c) shows the average link utilization when the length of control interval varies.

period. The remaining tenants (referred to as T6 to T10) have insufficient demands during the measurement period. Low-demanded tenants may initiate their flow transfers at any time during the measurement period. In this experiment we first fix the length of control interval as four seconds. Different settings are considered in Section 4.7.1.2.

Figure 4.7(a) plots the runtime core link bandwidth obtained by each tenant in a 10-second measurement period. During this period, QShare's tenant-queue binding algorithm assigns each of the tenants in T1 to T7 a dedicated queue on the core link; T8, T9 and T10 are served in a shared queue. When low-demanded tenants are inactive at the early stage, T1 through T5 fairly share the entire core link capacity. Later on, low-demanded tenants T6, T8, T9 and T10 become active. As T8, T9 and T10 are in the shared queue, they all obtain their guaranteed bandwidth. T1 to T6, each exclusively occupying a queue, equally share the remaining capacity. At about eight second, T7 becomes active and fairly shares the core link with T1 to T5. It is clear that all tenants receive at least their guaranteed bandwidth regardless of their communication patterns and other tenants' demands. Meanwhile, the core link is always fully utilized. Thus, QShare achieves perfect work-conserving bandwidth guarantees.

4.7.1.2 Unpredictable Demand Trend

In this section, we consider the case when tenant demand trend is unpredictable. We clarify that the predictability of traffic demand is only relevant to QShare's tenant-queue binding module, which affects the performance of work conservation. Thus, we mainly focus on the performance of work conservation when handling unpredictable demands.

We use the same set of tenants as in Section 4.7.1.1. To generate unpredictable traffic demands, each client requests flow transmissions from randomly selected servers. Flow sizes are sampled from the empirical datacenter workloads [141]. When the current flow finishes, a client randomly switches between being active (*i.e.,* requesting a new flow transmission) or dormant (*i.e.,* sleeping for a random period of time before requesting a new flow transfer). All transmissions are completely random and do not follow any specific probability distribution.

Figure 4.7(b) illustrates the runtime core link utilization over a one-minute measurement period. We measure the aggregated link utilization from all ten-

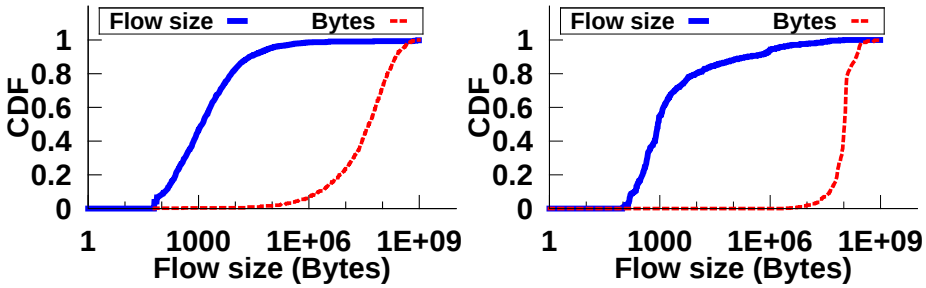
ants at the granularity of 0.1 second. As illustrated in Figure 4.7(b), in spite of unpredictable demands, under-utilized cases are rare, rendering over 91% average link utilization (plotted Figure 4.7(c)). This is because that QShare does not rely on good TM estimation to achieve work conservation. Instead, for any D-tenant (tenant with a dedicated queue), its VMs can burst traffic with arbitrary communication patterns, allowing them to effectively grab possible spare bandwidth. As long as one VM pair from all D-tenants is high-demanded, it can drive the core link to full utilization. Mathematically, the probability that all VM pairs from D-tenants have insufficient demands is low. In particular, assuming each VM pair independently determines to be either active or dormant with equal probability during a small time interval, the probability that the core link observes insufficient demands in the small interval[3] is $(\frac{1}{2})^{\mathcal{N}}$, where $\mathcal{N}$ is the number of VM pairs from all D-tenants. Thus, demand unpredictability has minor effects on work conservation.

We further plot the average core link utilization for different lengths of control interval in Figure 4.7(c). For predictable demand trend, QShare achieves perfect work conservation as long as the length of control interval is comparable with how long the trend lasts. For unpredictable trend, the utilization drops slightly as the length of control interval increases.

Based on this evaluation, we conclude that in order to achieve good work conservation, (i) QShare does not require perfect demand prediction and (ii) it is sufficient to perform tenant-queue allocation at coarse time granularity (*e.g.*, a few seconds). Thus, QShare's dynamic queue-tenant binding module does not need to react quickly enough to capture traffic bursts, which significantly reduces the stress for large scale deployment.

Fairness. We now consider the benefits of enabling ElasticSwitch-like rate allocations for the tenants without dedicated queues. First, it improves the link utilization for those under-utilized cases shown in Figure 4.7(b). Second, it improves the fairness for sharing the spare bandwidth since both tenants in the shared queue and tenants with dedicated queues are able to utilize such bandwidth.

[3]Given a small interval (*e.g.*, sub-millisecond), a small flow transmission may be considered as sufficient demand.



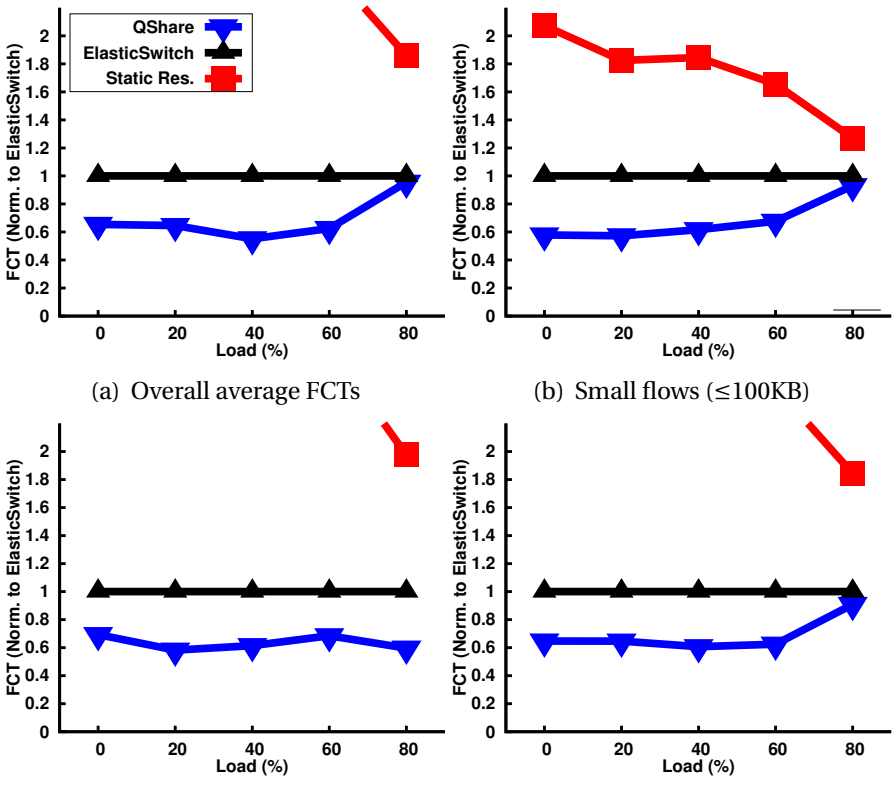
(a) Enterprise workload (b) Data-mining workload

Figure 4.8: Empirical traffic distributions used for measuring FCTs. The Bytes CDF shows the distribution of traffic bytes across different flow sizes.

4.7.2 Application Benefits

Given the desirable property in Section 4.7.1, QShare can benefit tenant applications by significantly reducing their flow completion times (FCTs). In this section, we demonstrate QShare's edges over ElasticSwitch [116], Trinity [120] as well as the static reservation for improving FCTs. Among all embedded tenants, we consider one tenant **T** with 10 VMs evenly distributed in two racks. Tenant **T** has 94 Mbps guaranteed bandwidth on the core link. We consider the shuffle phase of MapReduce jobs where a client requests flow transfers from all servers (recall that a VM runs both the client and server program). The flow sizes, illustrated in Figure 4.8, are sampled from empirically observed traffic patterns in two deployed datacenter traces [130] and [141]. Each client requests a new flow once the previous one is finished, indicating that **T** is high-demanded.

In the experiment, we create different datacenter fabric loads by varying the guaranteed bandwidth of background tenants (*i.e.,* the tenants competing with **T** on the core link). The load is computed as the ratio of total guaranteed bandwidth from background tenants to the core link capacity. The results for using the enterprise datacenter workload [141] are illustrated in Figure 4.9 (results for using the data-mining workload [130] are similar and we omit them for brevity). Because of the efficient resource utilization, QShare greatly reduces FCTs compared with both ElasticSwitch [116] and the static bandwidth reservation. Such improvement is even more significant for smaller fabric loads. In spite of its improvement over static reservation, ElasticSwitch [116] has a non-trivial performance degradation from QShare (up to 2x long FCTs) even if it adopts very aggressive RA to probe available bandwidth (scarifying band-



(a) Overall average FCTs

(b) Small flows (≤100KB)

(c) Medium flows

(d) Large flows (≥10MB)

Figure 4.9: FCT statics for varying fabric loads (part of the results for static reservation are out of the plot scope). Despite its improvement over static reservation, ElasticSwitch [116] has a large performance degradation compared with QShare (up to 2x long FCTs) even if it adopts aggressive RA at the expense of compromising bandwidth guarantees. In term of bandwidth utilization, Trinity has roughly the same performance as ElasticSwitch with aggressive RA.

width guarantees [116]). We are aware that ElasticSwitch's performance depends on parameter settings and system tuning. Our self-implemented ElasticSwitch prototype uses the default parameter setting in its paper. We do not further plot the results of Trinity [120] since ElasticSwitch with aggressive RA has roughly the same performance with Trinity in terms of bandwidth utilization (Section B), whereas Trinity has reordering and starvation issues.

We note that QShare is orthogonal to the approaches [4, 142–145] that reduce FCTs by creating more efficient transport protocols. Rather, QShare is an approach to allocate network resources for tenants.

4.7.3 QShare in Large Scale

In this section, we evaluate QShare in large scale. In particular, we shed light on the extent of switch queue scarcity (compared with the number of tenants) in large scale datacenters. Further, we show QShare's benefit for providing tenants more bandwidth than their guarantees and improving link utilization efficiency in large scale datacenters. We consider a three-layer multi-rooted tree topology with 1024 servers and 100 VMs per server, for a total of 100 thousand VMs. The network interface of each server is 10 Gbps and the switch port capacity is 40 Gbps. The network topology is constructed based on the $k = 16$ fattree [127] topology. By disabling certain links and switches, we can create a topology with different oversubscription ratios.

4.7.3.1 The Extent of Switch Queue Scarcity

To be consistent with the production datacenters [112, 118], the number of VMs requested by each tenant follows an exponential distribution with mean 49. The bandwidth guarantee of each VM is randomly sampled from five values 10 Mbps, 50 Mbps, 100 Mbps, 200 Mbps and 300 Mbps to better represent various bandwidth requirements from tenants. In the experiment, we keep embedding tenants until either network resources or computation resources are fully reserved, *i.e.,* the datacenter operates at 100% load. To do a stress test for queue scarcity, we assign more weight to c_q in Algorithm 2. We test three different over-subscription ratios $1:1$, $4:1$ and $16:1$.

The tenant placement results are tabulated in Table 4.1. Overall, the extent of

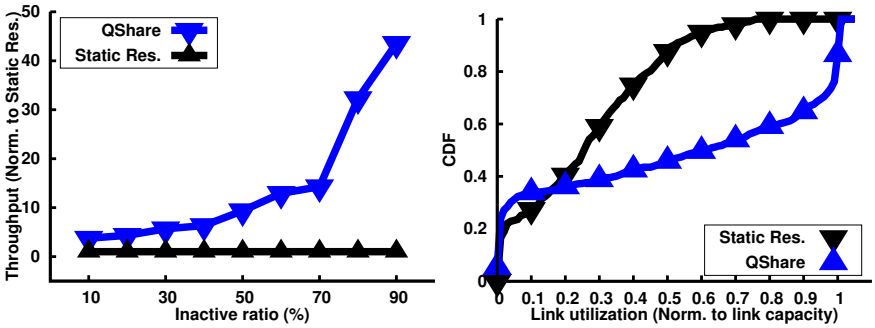
Table 4.1: Tenant placement results in a large scale datacenter. $\mathbf{R}_{\mathbf{N}_P<9}$ is the percentage of ports serving less than nine tenants. $\mathbf{R}_{\mathbf{N}_L\in[9,12]}$ and $\mathbf{R}_{\mathbf{N}_L>12}$ have similar definitions. $\mathbf{R}_{\mathbf{N}_D}$ is the percentage of tenants permanently assigned a dedicated queue and $\mathbf{R}_{\mathbf{N}_I}$ is the percentage of tenants assigned a dedicated queue, either permanently or opportunistically, in any control interval.

O. R.	$\mathbf{R}_{\mathbf{N}_L<9}$	$\mathbf{R}_{\mathbf{N}_L\in[9,12]}$	$\mathbf{R}_{\mathbf{N}_L>12}$	$\mathbf{R}_{\mathbf{N}_D}$	$\mathbf{R}_{\mathbf{N}_I}$
1 : 1	96.7	3.26	0	66.7	90.4
4 : 1	95.1	4.88	0	67.2	90.1
16 : 1	92.1	7.89	0	66.7	90.6

queue scarcity is moderate, counterintuitive to the common assumption [124]. For instance, only ~4% switch ports are overloaded in the 1:1 over-subscribed topology. Among the over-utilized ports, the largest number of tenants handled by a single port is 12, slightly higher than the total number of queues. From the tenants' perspective, two thirds of them are assigned dedicated queues throughout their lifetime due to the lack of queue contention, *i.e.,* on any link of their TRs, the number of competing tenants is fewer than eight. About 90% of all tenants can have dedicated queues, either permanently or opportunistically, in any control interval, indicating that only a small fraction of tenants need to run rate allocations at hypervisors. After placement, we assign tenants `dscp` values to analyze the `dscp` usage mentioned in Section 4.5.4. `dscp` zero is reserved for tenants in shared queues. For each tenant with dedicated queues, we greedily assign it the next non-conflicting `dscp` value. It turns out that 64 `dscp` values are sufficient even for the fully reserved datacenter.

We further emulate the processes of tenant arrival and departure. The tenant arrival is modeled by a Poisson process with rate λ and the lifetime of each tenant is a constant, similar to [121]. By varying λ, we tune the datacenter load. As the datacenter load drops, the queue scarcity is mitigated as well. When the load is less than ~60%, all tenants are permanently assigned dedicated queues. This demonstrates that our tenant placement module effectively spreads tenants across available switch queues to relieve queue contention.

The takeaway for this evaluation is that in reality, the problem of queue scarcity is moderate. By performing dynamic tenant-queue binding, QShare can effectively handle such scarcity in large scale datacenters.



(a) Throughput gain (b) Link utilization

Figure 4.10: QShare's performance in large scale. Figure 4.10(a) plots the high-demanded tenants' average throughput gain over the static reservation with various inactive ratios. Figure 4.10(b) shows the CDFs of normalized link utilization for QShare and static reservation.

4.7.3.2 QShare's Performance in Large Scale

In this section, we evaluate QShare's performance based on large scale simulations. We show that QShare produces significant throughput gain for tenants compared with the static reservation and achieves efficient link utilization in large scale datacenters. We develop a simulator incorporating QShare's tenant placement module and dynamic queue allocation algorithm.

The experiment is performed on the 16:1 over-subscribed topology as it is the most challenging setting in terms of queue scarcity. Meanwhile we still consider the tough scenario where the datacenter operates at full load, *i.e.,* resources are fully reserved. We define the inactive ratio r_{in} as the percentage of low-demanded tenants. Since the experiment tries to capture the tenant-level throughput, our simulator does not focus on detailed packet-level communications, but allowing the tenants to fully utilize the available bandwidth.

Throughput Gain. The throughput gain for a tenant is defined as the ratio of its actual achieved throughput to its guaranteed bandwidth. For simplicity, we assume the throughput gain for tenants in shared queues is one (no gain). For a tenant **T** with dedicated queues, its bandwidth gain on different links of its TR may be different since the actual demands on each link vary. We quantify the throughput gain of **T** as the smallest bandwidth gain obtained on any link of its TR. Thus, our experiment shows the worst-case throughput gain for **T** when the link with the smallest bandwidth gain is the bottleneck.

Figure 4.10(a) illustrates the average throughput gain given varying inactive

ratios. Overall, QShare produces significant throughput gains (*e.g.,* over 3x for all inactive ratios) over bandwidth guarantees. The throughput gain increases dramatically (up to ~50) as the inactive ratio increases, demonstrating that QShare can effectively utilize spare bandwidth.

Utilization Efficiency. A natural benefit of work conservation is that QShare can improve link utilization efficiency, *i.e.,* more links are operating at high utilization. Specifically, consider that tenant **T**'s throughput gain allows it to receive an extra 100 Mbps bandwidth besides its guaranteed bandwidth. This extra bandwidth will distribute among the links of **T**'s TR, driving these links to higher utilization. Without loss of generality, we consider a communication pattern spreading the throughput gain across **T**'s links proportionally to **T**'s guaranteed bandwidth on these links. As the throughput gain is obtained as the minimal bandwidth gain among all links, this distribution will not drive any link to over 100% utilization.

Figure 4.10(b) plots the CDFs of normalized link utilization (to the link capability) in the datacenter given $r_{in} = 0.5$. The results show that QShare achieves better efficiency in link utilization than static reservation. For instance, with QShare, half of the links' utilization is over ~60% compared with ~25% in static reservation; about 14% links are fully utilized with QShare compared with zero percentage in static reservation. These bottleneck links show that QShare has driven the network to the maximum possible utilization, *i.e.,* achieving work conservation.

4.7.4 System Properties

In this section, we report the following system properties to demonstrate QShare's scalability.

Switch Configuration. The network action container (Section 4.5.4) executes switch configuration commands in a batch. The latency for configuring queues on all 48 ports of our legacy switch is less than 50 ms, and the configurations on different switches are parallelized using multi-threading. For OpenFlow switches, configurations can be finished almost in real time via SDN controllers such as OpenDayLight [146]. Thus, even in large scale datacenter with thousands of switches, the overall configuration latency is negligible, compared with the length of control intervals (Section 4.7.1.2).

Table 4.2: Property comparison with related work. “BG" and “WC" mean bandwidth guarantees and work conservation, respectively.

	Oktopus [112]	Tag [121]	ES [116]	EyeQ [125]
BG	**Yes**	**Yes**	Tradeoff	**Yes**
WC	No	No	Tradeoff	**Yes**
Multi-tenant isolation & placement	No	**Yes**	No	**Yes**
Others	**None**	Application driven	**None**	Non-congested core

	Seawall [118]	Silo [5]	QJump [6]	TorPolice
BG	No	**Yes**	**Yes**	**Yes**
WC	**Yes**	No	No	**Yes**
Multi-tenant isolation & placement	No	**Yes**	No	**Yes**
Others	**None**	Hardware modification	Hardware modification	**None**

CPU Overhead. The major CPU overhead is contributed by the kernel module on hypervisors (Section 4.6), which is affected by traffic volume. At the full NIC speed (940 Mbps), we measure ~3% CPU overhead on our servers (shipped with a quad-core Intel 2.8 GHz CPU).

Tenant Placement. In the large-scale network topology in Section 4.7.3, the average time for figuring out the most desired TR for a tenant request is ~60 ms whereas the worst case takes no more than 100 ms.

4.8 Related Work

Table 4.2 summarizes the properties of some closely related work. SecondNet [117], Oktopus [112], and TIVC [147] provide static, non work-conserving bandwidth guarantees. EyeQ [125] and GateKeeper [126] achieve work-conserving bandwidth guarantees only if the network core is congestion-free, which may be not true for many datacenters [114, 115, 135]. ElasticSwitch [116] relies on challenging traffic matrix estimation and has a tradeoff between providing accurate bandwidth guarantees and being sufficiently work-conserving. Trinity [120] improves ElasticSwitch's work-conservation in static context via in-network priority queuing. However, it has starvation and packet reordering issues. Although Silo [5] and QJump [6] can provide both bandwidth and in-network latency guarantee, Silo is not work-conserving and QJump lacks the

tenant placement and isolation.

Using switch queues has been proposed before. For instance, vShaper [148] proposes to virtualize the physical queues to mimic the traffic shaping behavior of more queues, without considering bandwidth guarantees. pFabric [144], QJump [6] and PIAS [145] instead use priority queues to achieve low latency, although pFabric requires new hardware support, such as P4 [149].

FairCloud [124] proposes several models (or design principles) for sharing the network resources in datacenter. QShare's design follows the PS-P model, which, in theory, supports both work conservation and bandwidth guarantees simultaneously.

The bandwidth guarantees defined in the hose model can be enforced either at the level of per-tenant (*e.g.,* [112, 129]) or at the level of VM pairs, as proposed in [116, 120, 121]. Generally, QShare enforces per-tenant guarantees. However, for tenants without dedicated queues, their bandwidth guarantees need to be enforced through VM-pair guarantees.

4.9 Chapter Summary

This chapter presents QShare, the first complete in-network solution enabling work-conserving bandwidth guarantees in multi-tenant datacenters. At its core, QShare's tenant placement module provides accurate bandwidth guarantees, and its tenant-queue binding module dynamically assigns high-demand tenants dedicated switch queues to achieve work conservation. We implement a prototype of QShare, and perform extensive evaluations on physical testbed and via simulations to validate QShare's design goals. The results show that QShare improves state-of-the-art solutions in two aspects: (i) it does not rely on challenging traffic matrix prediction to achieve good performance and (ii) it eliminates the tradeoff of providing good bandwidth guarantees and being work conserving without raising starvation or packet reordering issues. Finally, QShare imposes small system overhead.

CHAPTER 5

MANAGING VIRTUAL NETWORKS IN MULTI-TENANT DATACENTERS: A SEARCH AND OPTIMIZATION PROBLEM

5.1 Introduction

Along with the rise of cloud computing, multi-tenant datacenters grow into the scale of thousands of switches and servers hosting tens of thousands of tenant Virtual Machines (VMs) [150, 151]. Managing multi-tenancy at such a scale to ensure efficiency, scalability and agility is a challenging problem drawing considerable research and engineering attention. Prior solutions [122, 152] place major management effort at hypervisors for the sake of easy configuration and implementation, whereas little effort has been made to manage *in-network* routings, particularly at the granularity of tenants. As a result, although the VM locations of each tenant are decided, datacenter network operators lose the visibility of actual in-network traffic forwarding for each tenant.

The lack of tenant-level traffic accountability and routing control could lead to various limitations. From the perspective of business, network operators cannot customize tenant routings in accord with tenants' service level agreements (SLA) (*e.g.,* latency, bandwidth, reliability or security requirements), which may close the door for such a business model in the virtual private cloud (VPC) market. From the perspective of management, in case of resolving hot spots in datacenters, for instance, it is difficult for network operators to determine the affected tenants and effectively re-route their traffic around congestion in time.

Thus, it is desirable to have explicit tenant routing control. The traditional way to achieve tenant routing management relies on topology search coupled with an objective function to greedily find the desired overlay network for each tenant. Although instant network configuration is technically enabled by SDN [153, 154], the conventional approach still has at least two shortcomings. First, tenant routing updates are all-time tasks, which can be triggered by vari-

ous reasons including network load dynamics, tenant arrivals/departures, hot spots, link failures and so on. Repeatedly performing topology search for each routing update will impose significant search cost. Second, a tenant routing is updated typically to fulfill certain goals. However, it is uncertain that topology search coupled with an objective function is sufficient to achieve any goal. For instance, optimizing a performance metric depending on virtual links (*i.e.*, VM pair communications) is subject to sub-optimality since the mapping between physical links and virtual links is unknown during topology search (Section 5.2.3).

To address these issues, we propose OpReduce in Chapter 5, a novel search and optimization *decoupled* design for routing management. For each tenant routing update, OpReduce first comprehensively searches all desired overlay candidates considering only the tenant's VM locations. Then it applies a global objective function over these candidates to finalize the most desired one. OpReduce's decoupled model offers at least two advantages. First, since topology search is not directed by any objective function, search results are general and can be reused across routing updates that share the same VM locations: *i.e.*, regardless of their goals, topology search for future routing updates is saved as long as their VM locations have been explored. Second, with the global view of all routing candidates, objective functions are not limited to be greedy and local. This eliminates the possibility of sub-optimality even when optimizing complex performance metrics, for instance, involving virtual links.

One concern of the decoupled design may be that finding all desired overlay candidates for a tenant can be expensive. However, since the VMs of one tenant are typically spanning across a few racks, we can reduce the search space to a subgraph of the entire topology. Further, most datacenter topologies (*e.g.*, VL2 [130], fattree [127]) are hierarchically organized into several layers and network traffic is never forwarded back and forth between different layers. We can adopt these properties to further refine the search space. Our proposed routing search algorithm imposes small complexity (Section 5.3.1).

We implement a prototype of OpReduce and perform extensive evaluations to validate its design goals. On the one hand, we show that OpReduce greatly reduces the search cost for tenant routing updates, meanwhile imposing small system overhead for managing large-scale datacenters (*e.g.,* small routing cache size and agile network configuration). On the other hand, we demonstrate that OpReduce is able to achieve complex optimization goals for routing

updates, yielding significant networking performance improvement over common practice.

5.2 Background and Motivation

5.2.1 Managing Per-Tenant Routing

For the sake of configuration simplicity, encapsulations protocols such as VXLAN [155] and NVGRE [156] are widely adopted in multi-tenant datacenters [152]. However, by simply tunneling tenant traffic, network operators lose the traffic visibility and accountability, *i.e.,* they are not able to identify the origin of network traffic since packets are sent on behalf of hypervisors and one hypervisor may host VMs for multiple tenants. Such invisibility becomes even worse when datacenters run link aggregation and/or perform load balancing (*i.e.,* ECMP, Hedera [157], or CONGA [141]) to spread traffic across redundant physical links. As a result, even if one tenant occupies only a small fraction of computation resources in the datacenter, its traffic could appear on many physical links, making network operators unaware of actual in-network traffic forwarding for the tenant.

The lack of in-network tenant routing management causes various operation restrictions. For instance, it is difficult for network operators to perform monitoring, measurement and trouble-shooting for a tenant as its traffic may spread across many network links. Further, as virtual private cloud gains popularity, more tenants have incentives to customize their private cloud based on their own needs, *e.g.,* one tenant may want to promote latency performance of web servers, whereas someone else may want to optimize bandwidth for MapReduce tasks. Without explicit tenant-level routing control, it is difficult to customize routings for individual tenants.

Thus, it is desirable for network operators to have explicit tenant routing control. In particular, for each tenant, network operators explicitly configure a Virtual Tenant Network (VTN), which is an overlay network connecting the tenant's VMs. The tenant's traffic is confined within its VTN and no other physical links besides the ones in its VTN can carry the tenant's traffic. As tenants can be identified by their VTNs, traffic is accountable, which eliminates these management limitations. Further, network operators can customize a tenant's routing

based on its requirement by embedding its VTN into an overlay that can best satisfy the requirement.

The traditional way of embedding VTN for a tenant relies on topology search coupled with an objective function to find the desired overlay. Such an approach, however, has at least two shortcomings: (i) it runs into scalability issues when handling frequent VTN embedding requests and (ii) it may be insufficient to find the optimal overlay for complex embedding goals. Next, we elaborate on the two issues.

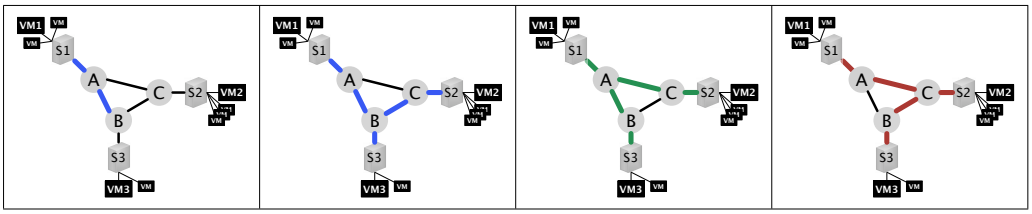
5.2.2 Frequent VTN Updating Requests

In multi-tenant datacenters, VTN embeddings/updates are all-time tasks on the basis of individual tenants, which are up to tens of thousands [150, 151]. VTN embedding requests are triggered by many sources including network load dynamics, link congestions and failures, hot spots, tenant departures and so on. Although efficient graph search algorithms, such as Prim's and Kruskal's, have been proposed for decades, consistently and frequently performing overlay search for a large number of VTN updates still imposes significant cost. Resolving the scalability problem is the first step towards efficient tenant routing management in large-scale datacenter networks.

5.2.3 VTN Embedding Goals

A VTN embedding/update may not be as simple as finding a random overlay network for the tenant. Instead, the VTN may need to satisfy various requirements such as guaranteed bandwidth [112, 117, 121], bounded latency [5, 6], required security appliances [158, 159], and other performance metrics depending on the tenant's SLA. As these embedding goals become more complex, it is uncertain that the conventional search and optimization coupled solution can find the real optimal routing.

We use a simple illustrative example in Figure 5.1 to demonstrate such insufficiency. Network operators perform VTN update for a tenant whose VMs are hosted by the set of hypervisors $[S_1, S_2, S_3]$. The embedding goal is to minimize the communication cost of the tenant's virtual links. For simplicity, we define the communication cost of a virtual link as the number of hops on the path



(a) Intermediate. (b) Default. (c) Alternative. (d) Optimal.

Figure 5.1: Due to the lack of mapping between virtual links and physical links, conventional search and optimization coupled solution is subject to sub-optimality when optimizing performance metrics depending on virtual links.

connecting the two VMs of the virtual link. Thus, if VM1 and VM2 take path S_1-A-C-S_2, the cost is three.

This embedding is a classic minimum spanning tree problem that can be solved by a greedy topology search algorithm. At certain intermediate state of the overlay search (Figure 5.1(a)), AB has been appended to the overlay. Based on the tie breaker in the algorithm, BC is then appended to the ongoing overlay, and the final overlay returned by the algorithm is S_1-A, S_2-C, S_3-B, AB, BC (Figure 5.1(b)). With a different algorithm, AC could be the link to append after state (a) rather than BC, and the final overlay returned will the one shown in Figure 5.1(c). However, no matter how the search algorithm is designed, the optimality of an overlay is up to the distribution of VMs. Although VM locations are known a priori, in an intermediate state, the search algorithm cannot determine exactly which virtual links a physical link will carry. Thus, the search algorithm cannot guarantee to capture the real optimal overlay (Figure 5.1(d)). In fact, due to the lack of mapping between virtual links and physical links, the conventional search and optimization coupled solution is subject to sub-optimality when optimizing any performance metric involving virtual links.

5.2.4 Decoupling Search and Optimization

To address the above challenges, OpReduce proposes a novel search and optimization decoupled design. For a routing update request, OpReduce first comprehensively finds all desired overlay candidates considering only VM locations. Then it applies an objective function, designed with the global view on all routing candidates, to finalize the optimal one. In the above example,

OpReduce will first obtain all three overlay candidates (figures (b), (c) and (d) in Figure 5.1) and then evaluate them to find the optimal one (Figure 5.1(d)). With the global view on all candidates, OpReduce knows the exact mapping between virtual links and physical links. Thus, it can design a global objective function in evaluation to guarantee optimality. Further, as topology search is not directed by any objective function, search results are general so that they can be reused for further VTN updates that have the same VM locations. This saves considerable cost for performing frequent VTN updates in large-scale datacenters.

5.3 System Design

In this section, we elaborate on the design of OpReduce. Figure 5.2 illustrates the architecture of OpReduce. OpReduce is built upon a network information database and a controller. The network information database allows OpReduce to retrieve network related information, such as network topology and link utilization. The controller is used to manage both computation and networking resources, such as assigning VMs to tenants and configuring VTNs for tenants. OpReduce's decoupled design is achieved by its VTN embedding module which includes routing search engine, routing cache, objective functions and network action container.

Next, we describe the workflow of performing routing update using OpReduce. Upon receiving a request, OpReduce first obtains a list of desired routing candidates based on the VM placement. These routing candidates may come from either the routing search engine or the routing cache if the VM placement has been explored before. Then OpReduce needs to evaluate each candidate to determine the most desired one for achieving an embedding goal. To assist evaluation, network operators propose evaluation methods, defined as objective functions, to score each routing candidate. The design of objective functions is facilitated by the global view of all possible routing candidates. Further, network information that is helpful for evaluation can be retrieved from the network information database. Finally, after determining the most desired routing, OpReduce enforces the VTN embedding inside the network by performing a list of network tasks via the network action container.

Although setting up the network information database, the network controller and the computation controller takes considerable implementation ef-

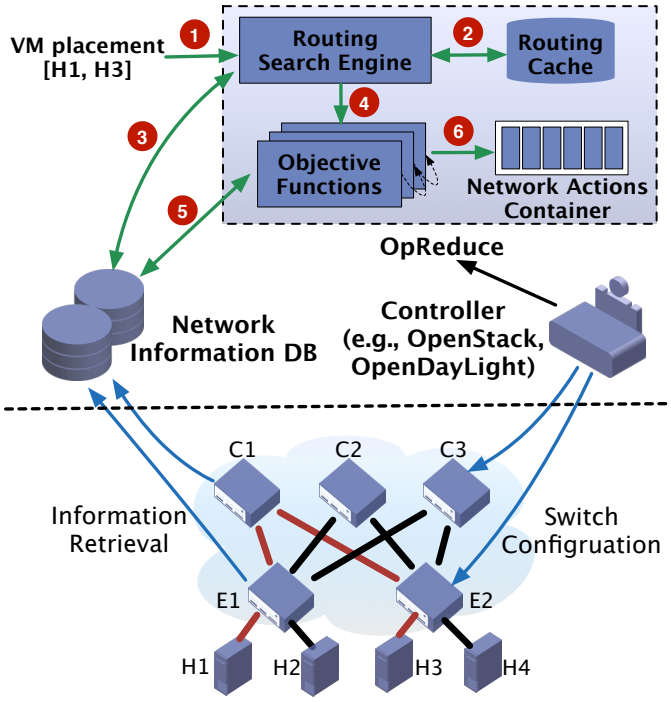


Figure 5.2: The architecture of OpReduce.

forts, we omit these details to focus on the research part of OpReduce in the thesis. In the following, we describe each individual component of OpReduce's VTN embedding module. Hereafter, we use *routing* and *VTN* interchangeably to indicate the overlay network carrying a tenant's traffic.

5.3.1 Routing Search Engine

Given a certain VM placement, the routing search engine is used to produce a list of *desired* routings (or routing candidates). In the typical layered datacenter network layout (*e.g.*, fattree [127], Clos [130, 160]), a routing is desired if traffic between two VMs does not bounce back and forth between two different layers. For random topology (*e.g.*, [132, 133]), we can bound the maximum number of hops on VM-pair paths to exclude undesired routings. As datacenter fabric is often built with a high level of redundancy, exploring all desired routings will provide sufficient candidates to achieve the embedding goal of the routing update request.

In this section, we detail an algorithm for searching all desired routings in layered datacenter topology that meets the following requirements. (i) The topol-

ogy consists of a set of isolated zones (or pods), which are inter-connected by several core switches. (ii) Each pod consists of a few layers. Switches on one layer are connected only to switches on different layers. Most of the common datacenter implementations such as VL2 [130] and fat-tree [127] satisfy these requirements.

In the network, we associate each node with a *height*. All core switches are assigned height 1 and the initial height for other nodes is infinity. Then we trace down from core switches toward other nodes to assign them a height. The height of a node is the minimum height among all its neighbours plus one. All nodes with the same height compose a layer whose layer number is the node height. We define a path as *straight* if all nodes on the path have different heights. Straight path is either upward (from a source at a lower layer to a destination at a higher layer) or downward.

Note that traffic in desired routings is never forwarded back and forth between different layers. Thus, to connect two hypervisors h_1 and h_3, h_1 needs to have a straight path which shares the same endpoint with one of h_3's straight paths. The shared endpoint is defined as a *common node* for the hypervisor set $[h_1, h_3]$. Formally, a node is defined as the common node for a set of hypervisors if it has at least one straight path to reach each hypervisor in the set. Then traffic from h_1 can first take one straight path to reach the common node and then bounces back to reach h_3 via another straight path. Thus, a routing connecting $[h_1, h_3]$ is composed of the above two straight paths. Similarly, a routing connecting all VMs of a tenant is composed of several straight paths, among which each straight path is originated from one hypervisor and all these paths are ended at one same common node.

We formulate our search procedure in Algorithm 4. At the very high level, the algorithm works as follows: (i) find all common nodes for the hypervisor set (line 5); (ii) for each common node, find one downward straight path from the common node to each hypervisor (line 8), and (iii) combine these paths to produce one routing candidate (line 9). Since a routing candidate can only contain one common node (having more will produce loops), Algorithm 4 provably finds all desired routings in the network. Even though the algorithm comprehensively finds all desired routings, the algorithm complexity is $O(|E|)$ (not exponential), where $|E|$ is the total number of edges in all upward graphs (line 4), which is much smaller than the number of edges in the entire network topology. We defer detailed complexity analysis in Appendix C.

Algorithm 4: Routing Search Algorithm

Input: VM placement H.
Output: All desired routing candidates.

---

for $h_i \in$ hypervisor set H **do**
    $\mathcal{T}_i \leftarrow$ GetUpwardGraph(h_i); $\mathcal{T} \leftarrow \bigcup_i \mathcal{T}_i$;
CommonNodes $\leftarrow \bigcap_{\mathcal{T}_i \in \mathcal{T}} \mathcal{T}_i$.nodes();
foreach $c_j \in$ CommonNodes **do**
    for $h_i \in$ hypervisor set H **do**
        $\mathcal{N}_j \leftarrow \bigcup_i$ GetDownwardPath(c_j, h_i, $\mathcal{T}_j$);
return $\mathcal{N} \leftarrow \bigcup_i \mathcal{N}_i$;

Function: GetUpwardGraph (h_i)
    return The graph containing all upward straight paths starting from h_i and ending at core switches.

Function: GetDownwardPath(c_j, h_i, $\mathcal{T}_j$)
    return The straight path from c_j to h_i in graph $\mathcal{T}_j$.

For a random datacenter topology [132, 133], OpReduce can first adopt the k-shortest path algorithm [134] to obtain a set of paths between each hypervisor pair and then combine these paths to produce routing candidates. The k can be parameterized to exclude undesired routings.

5.3.2 Routing Cache

The search results for routing candidates are cached using a dictionary (hash table) data structure, where the key is VM placement and the value is a list of all desired routings for the VM placement. Each routing is stored as a list of physical link IDs, which can be used to retrieve link related information (*e.g.,* utilization, status) from the network information database. Generally, entries in the dictionary are valid as long as the network topology remains the same. In OpReduce's prototype, we associate each entry in the dictionary with a relatively long validation period (*e.g.,* few weeks), and re-perform routing search after an entry is expired.

In OpReduce's implementation, we allocate 32 bits for both the hash key and physical link IDs. In Section 5.5.2.2, we show that even in large-scale $k = 32$ fat-tree datacenter with over eight thousand servers, the cache size for manag-

ing 10 thousand tenants is about hundreds of megabytes, which can be easily managed by commodity servers.

5.3.3 Objective Functions

The objective functions are used to evaluate routing candidates so as to determine the most desired one for fulfilling embedding goals. For instance, a valid objective function can be as sophisticated as a combination function balancing latency, bandwidth, the number of hops and so on. Or it can be as detailed as optimizing specific virtual links for latency and other virtual links for bandwidth. In general, OpReduce is open to accept any objective function. But OpReduce offers global views on all possible routing candidates so that objective functions are not limited to be greedy and local. Relevant network information, available in OpReduce's network information database, can be applied to evaluate these routing candidates. In production datacenters, to achieve a wide variety of VTN embedding goals, network operators can pre-install in OpReduce a set of abstract objective functions that can be parameterized accordingly.

5.3.4 Network Action Container

After determining the most desired routing, the final step is enforcing the routing inside network. The enforcement process is essentially configuring switches on the routing (*e.g.,* configuring VLAN tags [161] or adding Openflow rules [154]) to guide the switches to perform desired traffic forwarding. To facilitate routing management in large-scale datacenters, we develop a network action container to perform network configurations in a batch. For instance, configurations on different ports of one switch are aggregated in one thread and configurations on different switches are paralleled via multi-threading. As shown in Section 5.5.2.3, the network action container significantly reduces the overall configuration latency.

5.4 Implementation

We have a full implementation of OpReduce. We use OpenStack to create VMs on hypervisors and assign VMs to tenants. On our testbed dedicated for ex-

periments, the OpenStack environment has four compute nodes as hypervisors that can host about one hundred VMs, one network node, and one controller node. Because OpenStack is limited to computation virtualization, we unify it with our physical network via an OpenDayLight [146] controller that uses OpenFlow and `netconf` [162] protocols to manage OpenFlow and legacy switches, respectively.

On our testbed, we allocate a dedicate VLAN tag for each VTN. Then embedding a VTN is about configuring relevant switch interfaces with corresponding VLAN tags to achieve reachability. Certainly there could be other network virtualization solutions, *e.g.*, slices in sliceable switch [163], VLAN tag stacking [164]. OpReduce's network controller is extendible to support these virtualization solutions. We are aware that VLAN tags are limited to 4096 values which may be insufficient in large-scale datacenters. To resolve this scalability issue, OpReduce implements a helper component, based on Panopticon [165], to achieve VLAN tag reuse in hybrid datacenters with both legacy and OpenFlow switches.

5.5 Evaluation

Our evaluation centers around the followings. (i) In Section 5.5.1, we show OpReduce is guaranteed to find the least congested routings for VTN embeddings under various settings, which yields significant networking performance improvement over common practice. (ii) In Section 5.5.2, we show that OpReduce greatly reduces the search cost for managing numerous routing updates and imposes small system overhead for managing large-scale datacenters.

5.5.1 Congestion-Aware Routing Updates

One representative goal for routing updates in multi-tenant datacenters is to find the least congested routing to re-accommodate a tenant. The traditional search and optimization coupled solution for finding the desired routing is using the Prim's minimal spanning tree algorithm with physical link utilization as the edge weight. However, since network traffic is generated by VM pairs, minimizing the congestion experienced by virtual links is the more direct and therefore more accurate goal for finding the least congested routing. Thus, OpReduce uses the following algorithm to determine the most desired routing. Assume

OpReduce produces m routing candidates $\{\mathcal{T}_1, ..., \mathcal{T}_m\}$ for a routing update request that has n virtual links $\{l_1, ..., l_n\}$. Then the most desired routing is

$$\mathcal{T}^* = \min_{\mathcal{T}_k} \frac{\sum_{i=1}^{n} \lambda_i \cdot \mathcal{F}_k(l_i)}{n}, \tag{5.1}$$

where $k \in [1, m]$, $\mathcal{F}_k(l_i)$ is the congestion level experienced by virtual link l_i for routing candidate $\mathcal{T}_k$ and λ_i is the weight of l_i. Both $\mathcal{F}$ and λ_i are configurable to allow high flexibility for routing customization. For simplicity of performing evaluation in the thesis, $\mathcal{F}_k(l_i)$ is defined as the highest congestion level of all physical links in $\mathcal{T}_k$ that carry l_i, where the congestion level of a physical link is estimated as its average link utilization. Further, all virtual links are equally weighted.

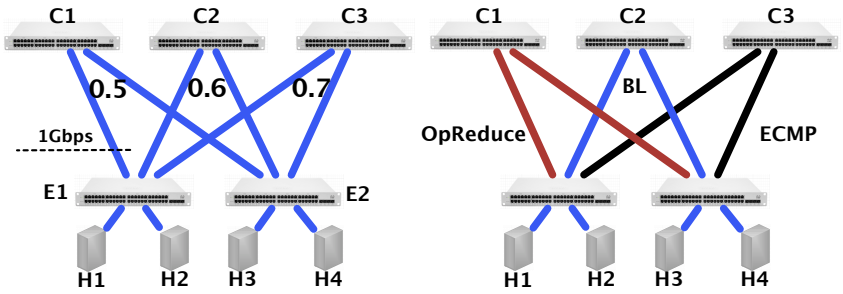
Besides the conventional search and optimization coupled solution (referred to as the *local* solution), we also compare OpReduce with flow-level ECMP, the common practice for load balancing and congestion reduction in datacenters, and the *bottomline* solution which embeds VTN to a randomly selected routing candidate.

For each embedding request, we compare the routing determined by OpReduce and the ones selected by the other three solutions. We use average flow completion time (FCT) as the metric to quantify the congestion experienced by virtual links (similar to [141, 166]), expecting that a better routing will have shorter average FCT.

5.5.1.1 Testbed Experiments

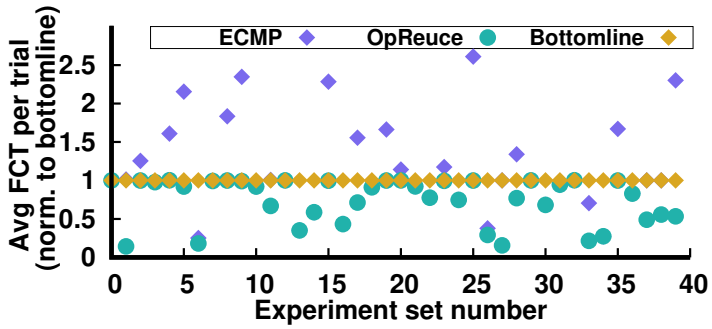
We start the evaluation on our physical testbed. We build a network topology illustrated in Figure 5.3(a). The datacenter has pre-embedded tenants to generate traffic using our client/server programs. The clients initiate long-lived TCP connections to randomly selected servers to request flow transfers. All VMs run both the client and server programs. Only intra-tenant communication is allowed.

In this experiment, we perform routing update for a tenant who has 10 VMs that are randomly distributed in all four hypervisors on our testbed. In total, we perform 40 sets of experiments under different network utilization. Figure 5.3(a) shows the snapshot of network utilization in one experiment set. In each experiment set, we perform three individual routing updates using OpReduce,



(a) Background utilization in one snapshot

(b) Desired routings produced by each solution



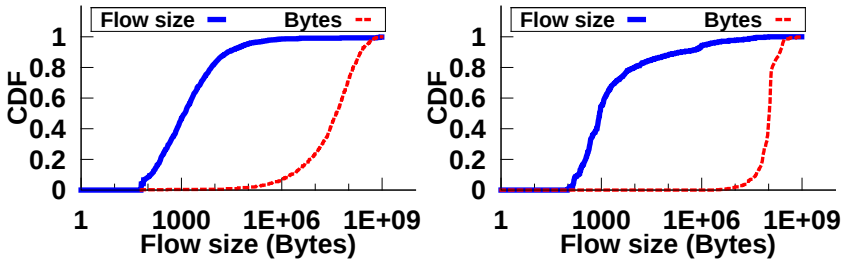
(c) Average FCT in each experiment set

Figure 5.3: Testbed experiments for improving FCTs.

ECMP and the bottomline solution. After the tenant VTN is embedded, its virtual links (VM pairs) start to generate traffic using our client/server programs. Flow sizes are randomly sampled from the empirical datacenter workload [141], whose flow size distribution is plotted in Figure 5.4. To ensure fair comparison, the set of flow sizes used for the three experiments in each experiment set is identical. We compute the average FCT among all flow transfers as our performance metric.

Figure 5.3(b) illustrates the desired routing produced by each solution for the network utilization shown in Figure 5.3(a). Since OpReduce first outputs all routing candidates (*i.e.*, E_1-C_1-E_2, E_1-C_2-E_2, E_1-C_3-E_2) and then applies the objective function (Equation (5.1)) to determine the most desired one, it can always find the least congested routing (E_1-C_1-E_2 in this case). However, both ECMP and bottomline are unaware of the link utilization. Consequently, their static hashing could result in overwhelming these more congested links.

Figure 5.3(c) plots the average FCT in each experiment set when using enterprise workload [141] (results for using data-mining workload [130] are similar, and we omit them for brevity). Although the bottomline solution and ECMP



(a) Enterprise workload [141] (b) Data-mining workload [130]

Figure 5.4: Empirical datacenter workloads used in evaluation.

may end up with different FCTs in one set, their average FCTs over 40 sets are close. Comparing OpReduce with the other two solutions, we find that OpReduce can consistently find the most desired routing which offers average ~30% less FCTs and up to ~5x reduced FCTs in some sets (outside the plot scope of Figure 5.3(c)).

Note we do not compare OpReduce with the local solution in our testbed experiments. This is because the local solution and OpReduce will produce the same routing in such a small topology. In the following, we show that given large network topologies, it is very likely for the local solution to return sub-optimal overlays, which results in significant performance degradation.

5.5.1.2 Large-Scale Simulations

In order to investigate how OpReduce's performance is affected by various factors that are not covered by our small-scale testbed experiments, we perform detailed simulations using large-scale datacenter topologies and empirical workloads obtained from production datacenters (Figure 5.4). As summarized in Table 5.1, we thoroughly evaluate the performance of OpReduce under the impact of five factors: the topology, the network/fabric load, the traffic workload, the average number of VMs occupied by a tenant, and the average VTN scale.

Various Fabric Loads (Exp NO. 1): In multi-tenant datacenters, the fabric load can be quantified as the VM over-subscription ratio, which is defined as the ratio of worst-case achievable aggregate bandwidth among VM pairs to the total capacity of the topology. A fat-tree fabric has an hypervisor over-subscription of 1:1 because all hypervisors may potentially send at the full bandwidth of their network interfaces. Thus, if the average number of VMs hosted by a hypervisor

Table 5.1: Experimental settings. "*" refers to that we vary the factor to isolate its impact in the experiment.

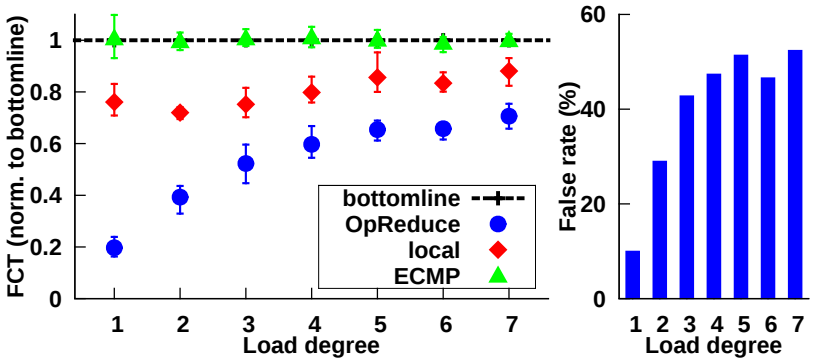
Exp. NO.	Topology	Fabric load	Workload	# VMs per tenant	Average VTN scale
1	k=8 FT	*	enterprise	20	4
2	k=8 FT	3:1	*	20	4
3	*	3:1	enterprise	20	4
4	k=8 FT	3:1	enterprise	20	*
5	k=8 FT	3:1	enterprise	*	4

is N, the VM over-subscription ratio is N:1. Hereafter, we refer the VM over-subscription ratio as the *load degree*.

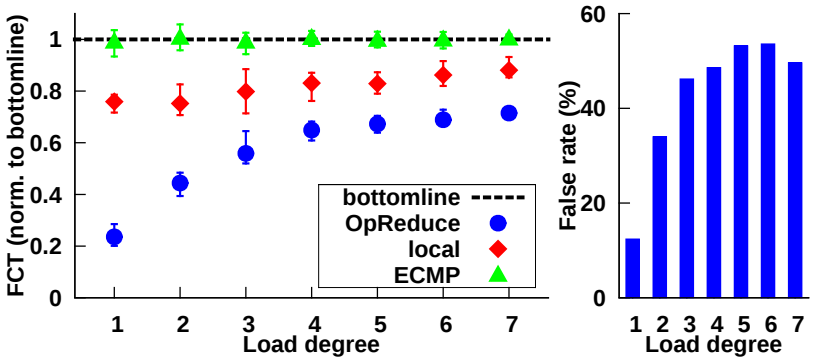
We tune the datacenter load degree in the range of 1 : 1-7 : 1 by varying the number of embedded tenants. For each load, we create five different snapshots to avoid bias. For each snapshot, we randomly pick a tenant to perform routing update using all four solutions: OpReduce, the local solution, ECMP and the bottomline solution. Thus, we perform four independent experiments for the tenant. In each solution, after the desired routing is finalized, we assign each virtual link a randomly sampled flow size to generate traffic and compute the average FCT among all flow transfers. Among all four experiments for the same tenant, we use the same set of flow sizes. After finishing the current tenant, we recover the network snapshot and re-sample another tenant to continue evaluation. In total, 100 routing updates are performed for each snapshot.

Figure 5.5 plots the min-median-max distribution of the averaged FCT across all five snapshots for each load degree. As illustrated in the Figure 5.5(a), OpReduce outperforms ECMP and bottomline with a significant margin: up to 80% FCT reduction for small fabric loads and at least 40% FCT reduction for all loads. Again, this is because ECMP and bottomline are static solutions that are unaware of the network utilization. Further, as load degree increases, OpReduce offers less FCT reductions since there is less routing optimization space in heavily utilized network. Since production datacenters typically have 3x over-provisioning to absorb traffic burstiness [167], we can expect large performance benefits offered by OpReduce in production datacenters.

Due to the lack of exact mapping between virtual links and physical links, the local solution often produces sub-optimal routings. As shown in Figure 5.5(b), with high probabilities (up to ~60%), the local solution returns an overlay different from the one produced by OpReduce. Consequently, these sub-optimal



(a) FCT min-mean-max distribution. (b) False rate.



(c) FCT min-mean-max distribution. (d) False rate.

Figure 5.5: Impact of the fabric load on OpReduce's performance. The normalized FCT is used as the performance metric. Parts (a) and (b) use the enterprise workload [141]. Parts (c) and (d) use the data-mining workload [130].

routings result in 1.2−4x FCT inflation, compared with the routings produced by OpReduce, as shown in Figure 5.5(a).

Datacenter Traffic Workload (Exp NO. 2): In Figures 5.5(a) and 5.5(b), we use the enterprise datacenter workload. We repeat the same experiment using the data-mining workload to learn the impact of traffic workload on OpReduce's performance. The results, plotted in Figures 5.5(c) and 5.5(d), show that OpReduce provide similar benefits for data-mining workload.

Fabric Topologies (Exp NO. 3): We investigate four datacenter topologies: three organized topologies (Clos, $k = 8$ and $k = 16$ fat-tree) and another organized topology added with random short-cuts. The Clos topology has the same number of hypervisors as the $k = 8$ fat-tree topology except that the oversubscription ratio is $2:1$. All three organized topologies have different routing redundancy. In particular, they have eight, 16 and 64 shortest paths between two randomly chosen hypervisors in different pods, respectively. The short-cut

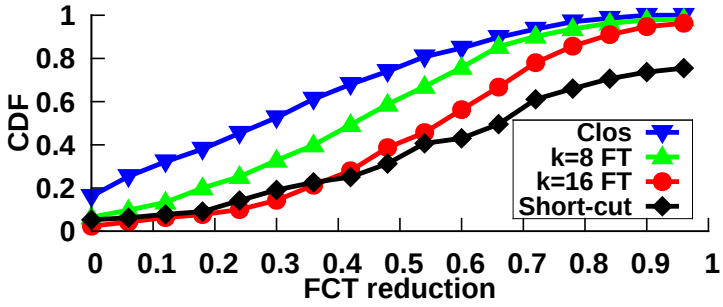
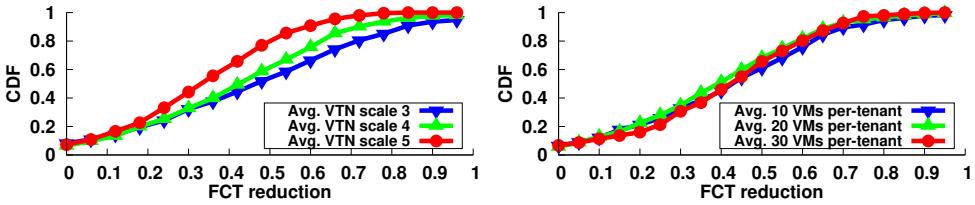


Figure 5.6: Impact of datacenter topology on OpReduce's performance of FCT reduction.



(a) Average VTN scale (Exp NO. 4)

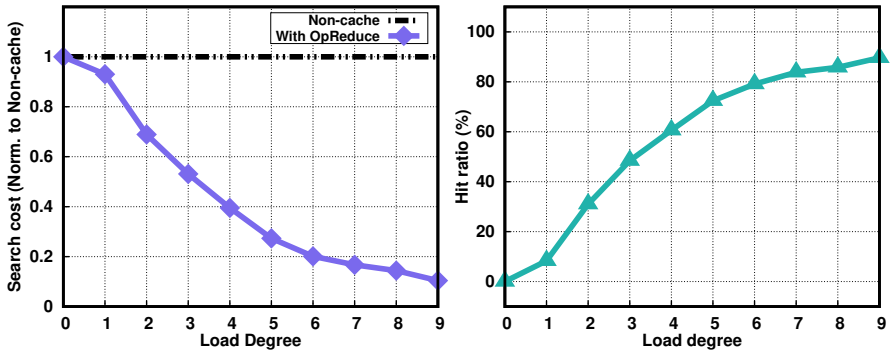
(b) Average per-tenant VM count (Exp NO. 5)

Figure 5.7: OpReduce offers consistent benefits for different VTN scales and different per-tenant VM counts.

topology is built by adding random links into the $k = 8$ fat-tree topology. The links are randomly added between ToR switches to inter-connect different pods so that besides traversing through the core switches, the inter-pod communication can alternatively use the short-cut bridges as well. We set the load degree as $3:1$ in all these topologies.

For each topology, we perform the similar experiments as in evaluating the impact of fabric loads. Figure 5.6 plots the CDF of all obtained FCT reductions (compared with the bottomline solution). By comparing the results of all three fat-tree topologies, we conclude that OpReduce offers more benefits for topologies built with more redundant links. This is because OpReduce has larger optimization space in more redundant topologies.

We further investigate how topology short-cuts may affect the performance of OpReduce. In organized datacenter topologies, alternative routing options typically have less diversity in the sense that they all have the same number of hops, although they are varying in terms of utilization. In contrast, short-cut topology can create more diverse routings with different numbers of hops as well as different utilization. Thus, OpReduce has even larger optimization



(a) Search cost reduction (b) Cache hit ratio

Figure 5.8: In spite of the numerous routing updates as load degree increases, OpReduce introduces small search cost since the cache hit ratio increases dramatically.

space in the short-cut topology so as to produce more performance benefits.

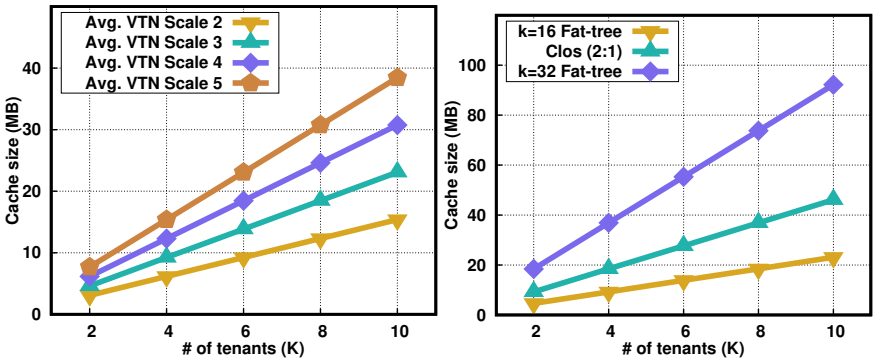
We also investigate the average VTN scale $\mathcal{N}_h$ (the average number of hypervisors used by a tenant's VMs) and the average per-tenant VM count $\mathcal{N}_v$. We consider $\mathcal{N}_h = [3,4,5]$ and $\mathcal{N}_v = [10,20,30]$ since they are the common practices in our production datacenters. We find that OpReduce consistently provides performance benefits in these settings, as shown in Figure 5.7.

5.5.2 System Properties

In this section, we present system evaluation.

5.5.2.1 Search Cost Reduction

The number of routing update requests is affected by the amount of embedded tenants, *i.e.,* the load degree. Figure 5.8(a) plots search cost reduction under various loads for $k = 16$ fat-tree topology. Although the absolute number of routing updates increase as load degree increases, the normalized search cost actually reduces. This is because OpReduce's knowledge base about routing candidates for various VM locations also increases as the load degree increases. As a result, the cache hit ratio dramatically increases as well (shown in Figure 5.8(b)). Thus, in spite of the numerous routing updates, OpReduce introduces small topology search cost.



(a) Various VTN scales (b) Various topologies

Figure 5.9: The routing cache size even in large-scale datacenters is small.

5.5.2.2 Routing Cache Size

The routing cache size is affected by two factors: the topology and the average VTN scale. The topology affects the cache size because it affects the number of routing candidates for one VM placement. The VTN scale affects the cache size since it determines the number of physical links in one routing candidate.

Figure 5.9(a) plots cache sizes with respect to the number of tenants and the average VTN scales in $k = 16$ fat-tree topology. The cache size increases with the average VTN scale and linearly grows with the number of tenants. However, even with very sporadically distributed VTNs, *i.e.,* large average VTN scales, caching routing candidates for 10 thousand tenants consumes no more than 40 MB memory. Even if we consider a much larger $k = 32$ fat-tree datacenter with over eight thousand servers, the cache size for 10 thousand tenants is about hundreds of Megabytes (Figure 5.9(b)), which can be easily managed by commodity servers with gigabytes of memory.

5.5.2.3 Switch Configuration Latency

Both SDN switches and legacy switches may co-exist in today's datacenters [168]. Configurations on OpenFlow Switches can be finished almost in real time via SDN controllers such as OpenDayLight [146], but it takes non-trivial time to configure legacy switches. Thus, to ensure that routing enforcement does not become the bottleneck for routing update, OpReduce designs a network action container to properly aggregate configuration tasks so as to reduce the overall configuration delay.

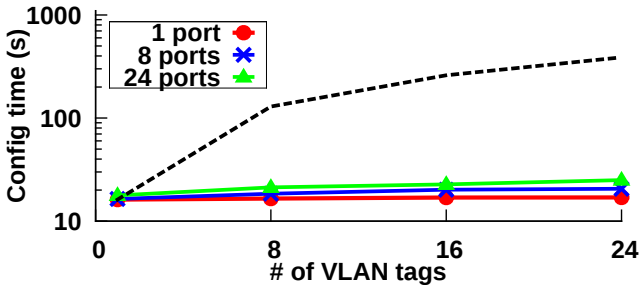


Figure 5.10: Delay measurements for configuring legacy switches. The dash line indicates the configuration time without using the network action container.

In OpReduce's prototype, one switch configuration task is about associating one VLAN tag on a certain port of a switch. We notice that configuring a single port on a legacy switch takes almost the same amount of time as configuring multiple ports on the switch. Thus, our network action container aggregates all configuration tasks on the same switch together to perform batch configuration so as to reduce the overall configuration delay. Meanwhile, batch configurations for different switches are executed simultaneously via multi-threading. Figure 5.10 plots the measured configuration delay on our testbed. The results show that even if we simultaneously configure 24 ports on one switch and on each port we configure 24 VLAN tags (576 single configuration tasks), the overall configuration time is less than 1.5 times the delay for configuring just one VLAN tag on a single port. Thus, with the network action container, routing enforcement can be finished timely so that it will not be a bottleneck in practice.

5.6 Related Work and Discussion

Multi-Tenancy Management. Prior designs for multi-tenant datacenters, such as NVP [152] and Netlord [122], focus on multi-tenancy management at hypervisors. For instance, NVP maintains virtual switches on each hypervisor and leverages a set of tunnels between each pair of hypervisors to deliver traffic for tenant VMs. The actual tenant traffic forwarding in the physical network is not managed. Several prior works have considered to perform one-time in-network routing management to achieve various goals, such as guaranteed bandwidth [112, 117, 121], bounded latency [5, 6] and user/service isolation [169]. OpReduce, on the other hand, focuses on managing routing updates, which is an

all-time task in our production datacenters.

A Wide Variety of Performance Enhancement. Many approaches have been proposed to improve datacenter networking performance. For instance, load balancing approaches [141, 157, 170, 171], priority queuing approaches [143, 144, 172], deadline-aware approaches [142] and DCTCP [4] are proposed to improve latency performance. Portland [173] and fat-tree [127] propose scalable datacenter architectures to support high bandwidth between servers whereas VL2 [130] virtualizes datacenters into server pools to allow applications to obtain high throughput. Although OpReduce is not proposed to explicitly improve certain performance, its efficient tenant routing management and decoupled design allow network operators to enhance a wide variety of *customer-interested* performance metrics, and some of these metrics cannot be optimized using prior approaches.

Achieving Agile Routing Updates. To be readily deployable in production datacenters, OpReduce is augmented by SDN to perform agile in-network routing updates. B4 [167] and SWAN [174] also adopt SDN to perform traffic engineering in wide area networks to achieve high inter-datacenter throughput.

5.7 Chapter Summary

This chapter presents OpReduce, a system for managing virtual tenant network update in multi-tenant datacenters. Conventional solutions that rely on topology search coupled with an objective function to find desired routings have at least two shortcomings: scalability issue for handling numerous routing updates and the inefficiency for satisfying various routing requirements. To address these issues, OpReduce proposes a novel search and optimization decoupled design, which enables routing search result reuse and guaranteed routing optimality. We implement a prototype of OpReduce and perform extensive evaluations to valid OpReduce's design goals. Evaluation results show that (i) even for complex VTN embedding goals, OpReduce ensures routing optimality which yields significant networking performance improvement over conventional approaches; (ii) OpReduce greatly reduces search cost for managing numerous routing updates and imposes small system overhead.

CHAPTER 6

CONCLUSION

The end-to-end principle plays a key role for the success of computer networks. However, over the past years, a number of new requirements have emerged for computer networks and their running applications, which creates various challenges in different networked systems. To address these challenges, there are no obvious solutions without adding in-network functions to the network core. Therefore, this thesis proposed the first design principle for guiding the implementation of in-network functions and then applied this principle to propose four designs in three different networked systems to address four separate challenges. Specifically, In the Internet, this thesis proposed MiddlePolice, the first readily deployable DDoS defense mechanism that can enforce a wide variety of victim-selectable policies during DDoS mitigation. In the Tor network, the thesis proposed TorPolice, the first privacy-preserving access control framework for Tor that enables service providers to effectively throttle Tor-emitted abuses and meanwhile serve legitimate clients. In multi-tenant datacenter networks, the thesis proposed QShare and OpReduce, in which QShare is designed to achieve work-conserving bandwidth guarantees and OpReduce is designed for optimizing tenant virtual networks based on tenant-desired metrics. For each proposed system, this thesis presented full implementations to demonstrate its feasibility in practice and performed extensive evaluations to validate its design goals.

APPENDIX A

TORPOLICE: TOWARD ENFORCING SERVICE-DEFINED ACCESS POLICIES FOR ANONYMITY SYSTEMS

A.1 Distributed Puzzle Systems

TorPolice introduces a distributed puzzle system for distributing computational puzzles. Compared with prior systems (*e.g.,* Portcullis [16]), the novelty of TorPolice's puzzle system is that it can explicitly bound the CPU usage by any client for solving puzzles. In particular, legitimate clients do not prefer to use all their CPU cycles to compute puzzles. However, automated bots do. To enable access control, prior systems (*e.g.,* Portcullis [16]) would need to prioritize requests based on the difficulty level of puzzles since otherwise the bots could overwhelm the system by solving easy puzzles. Thus, to compete with automated bots, legitimate clients are forced to use all their CPU cycles to solve puzzles while still at the risk of being denied access when facing automated bots with significant computation resources. On the contrary, by explicitly bounding the percentage of CPU cycles allowed for solving puzzles, TorPolice's can bring all bots down to the percentage that normal clients prefer to use for puzzle computation, which significantly reduces the computation disparity between legitimate clients and automated bots.

A.1.1 Puzzle System Overview

All computational puzzles are computed based on a series of *puzzle seeds* that are released periodically. Tor's existing directory authorities (DAs), for instance, can be used for releasing puzzle seeds. The puzzle system works on the basis of two periods, as illustrated in Figure A.1. In each *puzzle seed release period* (P_r^s), one fresh puzzle seed is released at the beginning of the period and no more puzzle seeds will be further released in this period. The seed release algorithm (Section A.1.2) ensures that the puzzle seeds cannot not be pre-computed and

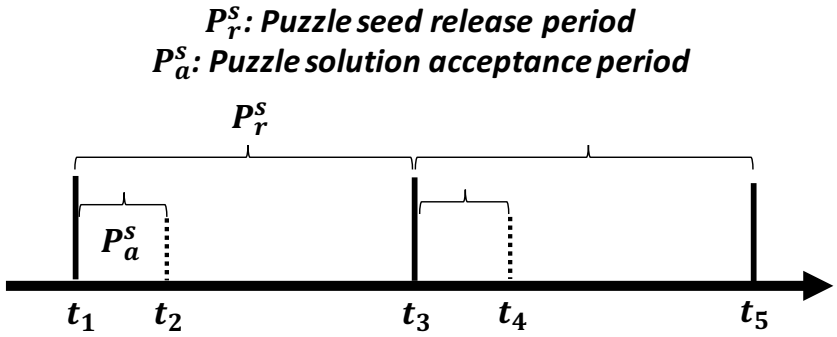


Figure A.1: TorPolice's puzzle system works on the basis of P_r^s and P_a^s: One fresh puzzle seed is released in each P_r^s and puzzle solutions are redeemable at AAs for pre-capabilities only within P_a^s in each P_r^s.

each valid seed requires the participance from a majority of all DAs.

In each P_r^s, all puzzles are computed based on the puzzle seed released in the current P_r^s. Thus, it is impossible to pre-compute solutions for future puzzles even if the puzzle generation algorithm (Section A.1.3) is public. Similarly, solutions to previous puzzles are unusable as valid capability seeds. To bound client's CPU usage for solving puzzles, all puzzle solutions have to be returned to the AAs within the *puzzle solution acceptance period (P_a^s)*. Late solutions will not be accepted. Thus, the percentage of CPU usage for solving puzzles is bounded by P_a^s/P_r^s.

A.1.2 Puzzle Seed Release

The puzzle seed release process requires the participation of at least n of Tor's DAs. n should include the majority of DAs to avoid centralization, and meanwhile it does not need to include all DAs to be fault-tolerant, similar to how the Tor network consensus is released. Specifically, each DA contributes its part for puzzle seed by generating a random nonce signed by its public key along with a timestamp, as formulated below.

$$s_i = n_i \mid t_s \mid \mathcal{S}_{D_i}, \tag{A.1}$$

where n_i is the nonce, t_s is the timestamp set to the starting time of the current P_r^s (*e.g.,* t_1 in Figure A.1) to indicate the freshness of the s_i, and $\mathcal{S}_{D_i}$ is the ith DA's signature to prove the integrity of s_i. To construct a puzzle seed for the current P_r^s, clients need to concatenate at least n authentic seed pieces issued

by n distinct DAs, We note that Tor's existing random value generator [76] does not fit for TorPolice since it computes a fresh value every day whereas TorPolice's puzzle seeds need to be released more frequently to improve usability, as explained in Section A.1.4.

A.1.3 Puzzle Computation

Assume that in the kth P_r^s, the puzzle seed obtained by a client is h_k. Then one puzzle is computed as follows.

$$p = H(p_{stub}) = H(h_k \mid r \mid s \mid \mathcal{F}), \tag{A.2}$$

where $H()$ is a public cryptographic hash function, r is a random 128-bit cryptographic nonce generated by the client to ensure the uniqueness of the puzzle, s is a 128-bit solution to the puzzle p, and $\mathcal{F}$ is the fingerprint of the AA selected by the client to redeem the puzzle solution. The s is considered as a valid solution to p only if $\frac{p}{2^{L_0-1}} < p_p$, where L_0 is the length of the hash function's output. p_p is a parameter for tuning the puzzle system. We provide detailed discussion for p_p in Section A.1.6. The $\{h_k \mid r \mid s \mid \mathcal{F}\}$ is defined as the puzzle stub p_{stub}, which will be sent to the AA (specified by $\mathcal{F}$) to redeem the puzzle solution. Incorporating $\mathcal{F}$ into the puzzle design prevents the client from redeeming a single puzzle solution at multiple AAs.

A.1.4 Puzzle Solution Acceptance Period

In order to be treated as valid capability seeds, puzzle stubs must be returned to the AAs within the puzzle solution acceptation period P_a^s, *i.e.,* an AA only accepts puzzle solutions received within $[t_1, t_2]$ in the current P_r^s (and equivalently $[t_3, t_4]$ in the next P_r^s), as illustrated in Figure A.1. The slot $[t_2, t_3]$ is the *cool-down* period, during which no puzzle stubs are accepted. As a result, a client, regardless of whether it is bot or a legitimate Tor user, can spend at most $\frac{P_a^s - P_c}{P_r^s} < \frac{P_a^s}{P_r^s}$ percent of its CPU cycles on solving computational puzzles, where P_c is the networking latency for retrieving the puzzle seed and returning the puzzle stub to the AAs.

Because of the cool-down period in each P_r^s, clients who missed the current P_a^s (either because they do not compute valid solutions on time or they obtain

the puzzle seed later than t_2) will have to wait until the starting of the next P_r^s (t_3) to get another chance to solve new puzzles. As such, P_r^s needs to be small (*e.g.,* at most few minutes) to avoid introducing usability problems.

A.1.5 Puzzle Solution Verification

To initiate puzzle verfication, a client sends the puzzle stub the corresponding AA. Upon the reception of puzzle stub, the AA performs the following checks to validate the puzzle stub. (i) The puzzle stub is returned within the current P_a^s. (ii) The puzzle stub is computed based on the fresh puzzle seed released in the current P_r^s. (iii) The puzzle stub encloses its fingerprint. (iv) The puzzle solution is valid, as defined in Section A.1.3. (v) The puzzle stub has not been spent before. To enforce the fifth rule, the AA needs to cache all spent puzzle stubs. The cache space is bounded as the AA can erase the puzzle stubs received in previous periods since they are no longer spendable.

A.1.6 Puzzle System Analysis

In each P_r^s, the number of puzzles solved by a client follows the following binomial distribution

$$\mathcal{G}_{seed} \sim \mathcal{B}\Big(p_p, \Big\lfloor \frac{P_a^s - P_c}{t_p} \Big\rfloor\Big), \tag{A.3}$$

where $\mathcal{G}_{seed}$ denotes number of solved puzzles, p_p is the probability that one attempt (*i.e.,* hash computation) produces a valid puzzle solution according to the rule in Section A.1.3, t_p is the amount of time it takes for the client to attempt a single hash computation and $\lfloor \frac{P_a^s - P_c}{t_p} \rfloor$ is the number of attempts $\mathbb{U}$ can make within the time limit.

TorPolice can control p_p and P_a^s to affect the numeric values of $\mathcal{G}_{seed}$. In particular, p_p should be chosen such that with high probability (*i.e.,* 0.99) a client with slow computation speed (*e.g.,* a mobile device released few years ago) and slow network connection (*e.g.,* 99th percentile of the RTT measured by CAIDA [175]) can correctly solve one puzzle so as to produce a valid capability seed. In particular, given that the slow device's computation speed is t_p^0 and the 99th percentile network latency is P_c^{99th}, p_p is selected such as $1-(1-p_p)^{N_0} > 0.99$, where $N_0 = \Big\lfloor \frac{P_a^s - P_c^{99th}}{t_p^0} \Big\rfloor$.

A.2 Trans-Capability Design Detail

In this section, we detail the capability exchange protocol introduced in Section 3.6.2. Since a Tor hidden server (itself runs a Tor client) needs to open many Tor circuits in order to serve all its clients (*i.e.,* HS-clients), enforcing per-seed rate limiting for pre-capability release may limit the availability of Tor HSes. To address this issue, we design the following capability exchange protocol.

In particular, a HS-client needs to request a new type of capability, *i.e., trans-capability*, from the AAs. During the hidden service set up process, along with the information about Rendezvous Point, the HS-client sends a trans-capability to one of the HS's Introduction Points. The HS subsequently redeems the trans-capability at the AAs for new pre-capabilities, which can be used for generating new relay-specific capabilities. The trans-capability, accounted on the capability seed of the HS-client, anonymously informs the AAs that the HS needs to create a new circuit to serve the HS-client.

Trans-Capability Computation. Each trans-capability is computed based on *pre-trans* issued by the AAs. By default, all Tor clients request pre-trans to prevent the AAs from knowing whether a client has the intention to visit HSes. Clients that do not visit any HS ignore the received pre-trans. To request pre-trans, the HS-client sends $\{\varsigma \mid n \mid t_s\}^b$ to the AAs, where ς is a pre-defined system value for trans-capability. No information about the HS is enclosed to protect the HS's privacy. The AAs then compute blind signatures over the information to produce a pre-trans. Finally, the HS-client unblinds the received pre-trans to produce a trans-capability.

Redeeming Trans-Capability. The process of redeeming trans-capability is identical to how a Tor client requests relay-specific pre-capabilities using its capability seed (the trans-capability now serves as a new capability seed), except that the HS contacts the AAs through Tor to hide its network location. The AAs reject all unauthentic, expired or spent trans-capabilities.

A.3 Live Tor Interaction

In this section, we continue our discussion in Section 3.8.4 for live Tor network interactive.

RelayManager Design. Since the live Tor relays are capability-agnostic (*i.e.,*

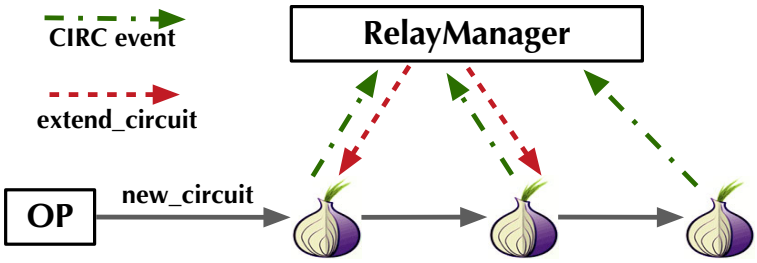


Figure A.2: The design of RelayManager.

they do not run our modified Tor source code described in Section 3.8.4), we cannot create TorPolice-enhanced Tor circuits directly through live Tor relays. Thus, we implement another prototype to interact with live Tor relays during capability-enhanced circuit creation. The prototype relies on the Tor control protocol [176]. In particular, on the OP, we implement a RelayManager based on the Stem [177] library to execute the capability-related operations, as illustrated in Figure A.2. The RelayManager controls the OP's circuit creation to "embed" capabilities into live Tor circuit creations. In particular, after the OP selects relays for its circuit. the RelayManager blinds relay information, requests pre-capabilities from our deployed AAs described in Section 3.8.2, and then computes relay-specific capabilities. Whenever the OP's partially-built circuit reaches a relay $\mathbb{R}_n$ in the live Tor network, the RelayManager receives a CIRC event callback from the Tor control protocol. As $\mathbb{R}_n$ is capability-agnostic, we offload capability verification to the RelayManager. Upon validation, the RelayManager sends an extend_circuit command through the Tor control protocol to continue circuit creation. Otherwise, the RelayManager terminates circuit creation by issuing a close_circuit command.

We clarify that RelayManager should not be used in real-world development since the capability verification is offloaded to the OP. Rather, the Tor relay source code needs to be modified to securely embrace TorPolice, as proposed in Section 3.8.4.

Latency Measurement. To validate the design of RelayManager, we instruct our Tor clients to create circuits through live Tor relays and use RelayManager to embed our capability-related operations into the creation process. Meanwhile, we measure circuit creation latencies to quantify the overhead caused by the our capability design. Figure A.3 plots the CDF of measured latency with and without relay-specific capabilities. The results show that TorPolice introduces

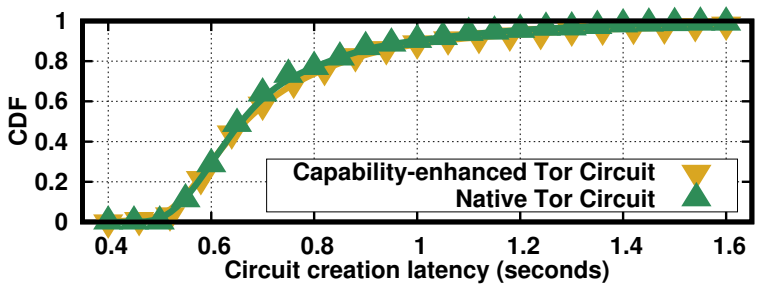


Figure A.3: TorPolice introduces negligible overhead for Tor circuit creation.

negligible overhead. This is because a capability verification operation merely takes ~0.03ms (Section 3.9.1) whereas the median time for creating a (native) Tor circuit is ~0.7s. We clarify that since the capability verification is offloaded from live Tor relays to the OP, there could be some marginal errors in our latency measurements since live Tor relays may simultaneously process multiple circuits requests.

A.4 Enforcing Site-Defined Policies

In this section, we continue the analysis in Section 3.9.3 to prove that an adversary's service request rate r_a is always bounded by $\Theta(\epsilon)$. In particular, when $k \leq 1$, CAPTCHA solutions are the optimal seeds since $c'_0 < c'_1$. Therefore, the optimal r_a is obtained when the adversary spends all investment on purchasing CAPTCHA solutions. Thus we have $r_a = \epsilon$. When $k \geq 1$, solutions to computational puzzles become the optimal seeds. In this case, the adversary's optimal r_a is $k \cdot \epsilon$ which is obtained by investing all money on solving puzzles.

When the site adopts the basic strategy (*i.e.,* accepts all Tor requests with valid capabilities), the adversary can continuously use optimal capability seeds to maintain its optimal r_a as its investment increases. However, if either the rate limiting strategy or the WFQ strategy is adopted, the adversary will reach a point of diminishing returns when the site no longer accepts service requests using capabilities that are obtained via the optimal seeds. As a result, the adversary's r_a starts to drop from the optimal value.

Thus, regardless of k and the site's strategy, r_a is always bounded by $\Theta(\epsilon)$. The above analysis can be easily extended to more types of capability seeds.

A.5 Modeling for Botnet C&C Abuse

In this section, we continue the discussion in Section 3.9.4.1 to detail the modeling for estimating the amount of circuit creation requests in Tor when Tor was under the large scale C&C abuse.

We collect the Tor network consensus published from September 1 to September 30, 2013 when the number of estimated daily Tor users ranged from four million to six million. Since one consensus file is published in each hour, we use the average statistics from all 24 consensus files published in a day to represent the Tor status in that day. We model the relay computation capacity based on the live Tor relay measurements in [69] by uniformly sampling their measurement numbers, excluding the samples with low confidence.

The number of circuit creation requests received by Tor is modeled by a Poisson Process with arrival rate λ. To compute λ, we first estimate the number of unique Tor clients in a time interval and then estimate the number of circuits opened by each client in the same interval. Mathematically, we have $\lambda = \frac{N_1 \cdot r_1 + N_2 \cdot r_2}{t_0}$, where N_1 and N_2 are the number of unique legitimate clients and bot clients, respectively, over the time interval t_0; r_1 and r_2 are the average number of circuit creations requested by a legitimate client and bot client, respectively, over the same interval t_0. N_1 and N_2 can be estimated using the metric inferred from live Tor measurements in [79]. In particular, over a 10-minute interval, PrivCount [79] counts 710 unique clients when Tor's daily estimated user count is 1.75 million, which indicates the client population *turnover rate* ρ is about 2.5. Since the methodology used by Tor to estimate its daily user has not changed since 2013, we assume that ρ obtained in 2016 is also applicable in 2013. Thus, over a 10-minute interval, we have $N_1 = N_1^L/\rho$ and $N_2 = (N_2^L - N_1^L)/\rho$, where N_1^L and N_2^L are the number of legitimate daily users and total daily users estimated by Tor, respectively. We estimate N_1^L as one million, which was the estimated daily Tor user number right before the abuse started in August 2013. Further, PrivCount [79] counts about four circuits opened for each Tor client over a 10-minute interval, thus we estimate r_1 is about 4 in a 10-minute interval, assuming that legitimate Tor clients had the same usage pattern in 2013 as they have in 2016.

However, the above usage pattern inferred from [79] cannot be applied to determine r_2 since bot clients may have different usage patterns from legitimate clients. Thus, we estimate r_2 using historical data. In particular, we find that

the highest circuit creation failure rate on September 27 2013 is about 35% [68]. Then using the network consensus of the same day, r_2 is estimated at about 150 over a 10-minute interval.

Based on the above modeling, we study the circuit creation failure rates using our Tor-scale simulator. The results are plotted in Figure 3.6.

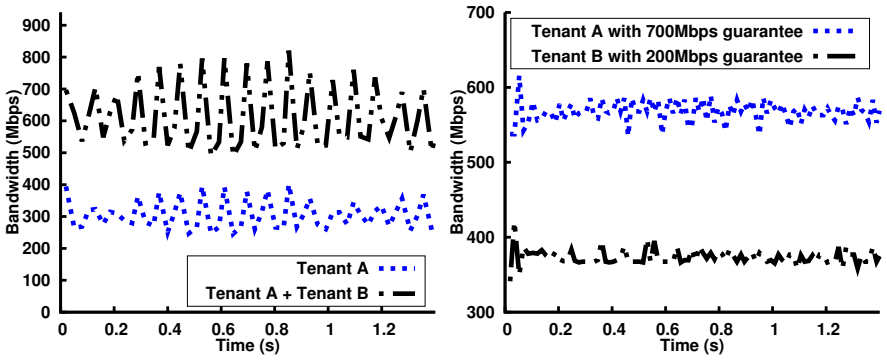
APPENDIX B

QSHARE: ENABLING WORK-CONSERVING BANDWIDTH GUARANTEES FOR MULTI-TENANT DATACENTERS VIA DYNAMIC TENANT-QUEUE BINDING

In this appendix, we revisit the experiments in Section 4.3.1. We hope to share our experience of experimenting with ElasticSwitch [116] and Trinity [120] to validate our motivation. We implement a prototype of both ElasticSwitch and Trinity based on the designs in their papers.

We first quantify ElasticSwitch's tradeoff of providing accurate bandwidth guarantees and being sufficiently work-conserving. Since the traffic matrix is unknown a priori, ElasticSwitch needs to allocate the bandwidth to each VM pair based on network condition probing. As a result, conservative allocation may result in bandwidth waste, especially when the total guaranteed (reserved) bandwidth is smaller than the link capability, whereas aggressive allocation may affect other tenants' guarantees, especially when large numbers of VM pairs are competing one congested link.

Conservative Allocation. During conservative allocation, ElasticSwitch [116] uses the following three mechanisms: (i) Headroom: leaving a gap between the link capacity and the maximum offered guarantees on any link; (ii) Hold-Increase (HI): delaying the rate increase after each congestion event and (iii) Rate-Caution (RC): being less aggressive to increase rates once the current rates are above the guarantees. Please refer to [116] for the details of each mechanism. We use the following experiment to show the bandwidth waste caused by the conservative allocation. Consider the case where both tenant A and B adopt the same symmetric hose model, in which each VM is guaranteed 50 Mbps bandwidth. Thus, both tenants have 250 Mbps guarantees on the core link, *i.e.,* the network link is half reserved. To generate traffic, each VM in one rack is configured to communicate with randomly selected VMs in the other rack, using our client/server program described in Section 4.7. Each VM's demand is completely random. Only intra-tenant communication is considered. We measure the total amount of core-link bandwidth utilized by each tenant. As illustrated in Figure B.1(a), we notice a significant gap (over 300 Mbps) be-



(a) Conservative probing (b) Aggressive probing

Figure B.1: Quantify the tradeoff between providing accurate bandwidth guarantees and being sufficiently work-conserving for ElasticSwitch [116].

tween the aggregated bandwidth of A and B and the link capacity, *i.e.*, over 60% of the unreserved bandwidth is wasted.

Aggressive Allocation. To achieve aggressive allocation, we disable all those three mechanisms used in conservative allocation so that the RA module immediately increases rates on each positive feedback (lack of congestion). We consider a case where tenant A has 700 Mbps guarantee and B 200 Mbps guarantee on the core link. As shown in Figure B.1(b), ElasticSwitch fails to guarantee A's bandwidth although it drives the link to higher utilization with aggressive allocation. In fact, ElasticSwitch [116] has demonstrated that bandwidth guarantees will be compromised once disabling RC and HI.

We are aware that ElasticSwitch's performance depends on parameter choices. Thus, in practice, network operators can boost ElasticSwitch's performance via system tuning. Our implementation uses the parameters specified in its paper.

Analysis. The causes of the above performance degradation are twofold: (i) the challenge of guarantees partitioning (GP) without prior knowledge of communication patterns and VM-pair demand, and (ii) the challenge of relying congestion feedback to learn real-time network bandwidth. Trinity [120] is effective to resolve the second cause since it does not need to learn the spare network bandwidth. Instead, it serves bandwidth-guarantee traffic in a prioritized queue so that senders can aggressively send work-conservation traffic without worrying that such aggressiveness would affect other tenants' guarantees. Thus, in terms of achieving high link utilization, Trinity and ElasticSwitch with aggressive RA have roughly the same performance. Both our experiments and

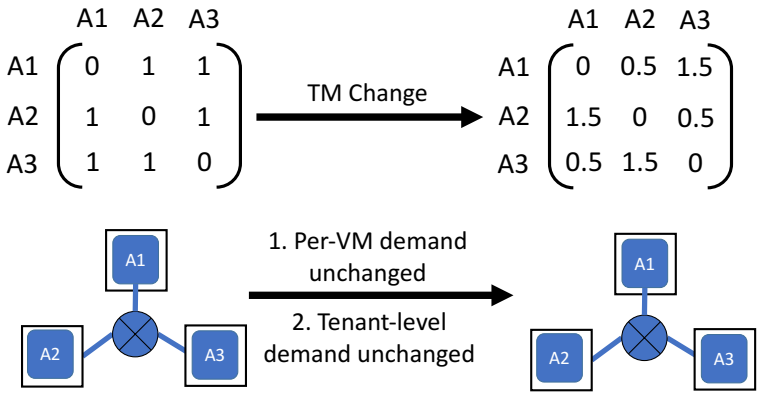


Figure B.2: The GP has to re-learn each VM pair's guarantee each time the TM changes, even when both per-VM and tenant-level demand remains the same.

its paper show that Trinity achieves good work-conservation in static context (the traffic matrix is known and stable). However, we also notice some practical issues such as starvation and packet reordering in our experiments with Trinity.

However, achieving good GP without prior knowledge still remains as an open problem. Since GP transforms each tenant's per-VM bandwidth guarantee defined in the hose model into traffic matrix (TM), it essentially complicates the model. Consider the illustrative example in Figure B.2. The item a_{ij} in the TM represents VM i's sending demand to VM j. When the actual TM changes from the left pattern to the right one, GP needs to gradually learn the update via probing. However, the demand of each VM actually remains the same despite the TM change. Further, given the VM placement in Figure B.2 (three VMs are placed in three different hypervisors), the amount of bandwidth required for the tenant on each link also remains the same. Therefore, GP needs to do extra work even when per-VM and tenant-level demand remain the same, which essentially increases the stress of traffic demand prediction.

Different from state-of-the-art solutions ElasticSwitch and Trinity [116, 120], QShare does not rely on GP in its design. Although QShare's tenant-queue binding module does require demand prediction, it is much more lightweight than TM estimation since QShare only predicts a scalar metric for each tenant. Additionally, as shown in our evaluations (Section 4.7.1), QShare does not require perfect prediction in order to achieve good work conserving bandwidth guarantees.

APPENDIX C

OPREDUCE: MANAGING VIRTUAL NETWORKS IN MULTI-TENANT DATACENTERS: A SEARCH AND OPTIMIZATION PROBLEM

In this appendix, we analyze the complexity of Algorithm 4 in Section 5.3.1. The first step of the algorithm is to find the upward graph for each hypervisor in H. Essentially, it is a breadth-first search process. Thus the search complexity is $O(|E|)$ where $|E|$ is the number of edges in the upward graph. Note that the upward graph is much smaller than the entire network. Specifically, $|E| = \sum_{k=2}^{L} R_k \cdot N_k$, where R_k denotes the link redundancy ratio at layer k, N_k is the number of nodes at layer k in the upward graph and L is the number of layers in the upward graph. The link redundancy ratio at layer k is defined as the number of links that one node in layer k can has to reach nodes in layer $k-1$. For instance, in $k = 8$ fat-tree topology with four layers, $R_4 = 1, R_3 = R_2 = 4$. There is no R_1 since layer 0 does not exist. Thus $|E| = 21$, which is very small compared with entire network size ($|E| = 384$). For simplicity of presentation, we assume all nodes in layer k have the same R_k. However, both our search algorithm and complexity analysis are not restricted by this assumption.

The second search step is that in upward graph $\mathcal{T}_j$, finding the downward straight paths from the common node to hypervisor h_i. It is depth-first search, which has $O(|E|)$ worst-case complexity. However, in topologies like fat-tree and Clos [160], $\mathcal{T}_j$ turns out to be a tree rooted at h_i with all the common nodes as its leaves. Thus only $L-1$ edges need to visited to find a downward straight path from the common node (leaf) to h_i (root), which introduces constant (negligible) overhead.

Table C.1 summarizes the search cost (the number of visited edges) for finding all desired routing candidates in different fat-tree topologies. We list the number of nodes and edges in the network to show that the search space is much smaller. $\mathcal{N}_h$ is the average number of hypervisors used by one tenant. We consider the complexity for both intra-pod (all the hypervisors are located within one pod) and inter-pod (the tenant spreads across at least two pods). For the special case that the all hypervisors in H are sharing the same ToR

Table C.1: One-time comprehensive search cost for fat-tree topologies. $\mathcal{N}_h$ is the average VTN scale.

Topo.	**Intra-Pod**	**Inter-Pod**	**(nodes,edges)**
k=8	$5\mathcal{N}_h$	$21\mathcal{N}_h$	(200, 384)
k=16	$9\mathcal{N}_h$	$71\mathcal{N}_h$	(1296, 3072)
k=32	$17\mathcal{N}_h$	$273\mathcal{N}_h$	(9248, 24567)
k	$(1+\frac{k}{2})\cdot\mathcal{N}_h$	$\left(1+\frac{k}{2}+(\frac{k}{2})^2\right)\cdot\mathcal{N}_h$	$(\sim\frac{k^3}{4}, \frac{3k^3}{4})$

switches, there will only one routing connecting them. It is clear that even in large-scale datacenters with thousands of servers, the one-time comprehensive search cost is small and acceptable. For other topologies with larger over-subscription ratio (*e.g.,* 2:1 Clos topology) than the fat-tree topology (1:1), search cost will further be reduced due to the smaller number of redundant paths in these topologies.

REFERENCES

[1] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems (TOCS)*, 1984.

[2] M. Prince, "The trouble with Tor," https://blog.cloudflare.com/the-trouble-with-tor/, 2016.

[3] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with Orchestra," in *ACM SIGCOMM*, 2011.

[4] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," *ACM SIGCOMM CCR*, 2011.

[5] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable message latency in the cloud," in *ACM SIGCOMM*, 2015.

[6] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues don't matter when you can jump them!" in *USENIX NSDI*, 2015.

[7] T. Anderson, T. Roscoe, and D. Wetherall, "Preventing Internet denial-of-service with capabilities," *ACM SIGCOMM CCR*, 2004.

[8] A. Networks, "Worldwide infrastructure security report, volume IX," https://www.arbornetworks.com/images/documents/WISR2016_EN_Web.pdf, 2016.

[9] T. T. Miu, A. K. Hui, W. Lee, D. X. Luo, A. K. Chung, and J. W. Wong, "Universal DDoS mitigation bypass," in *Black Hat USA*, 2013.

[10] T. Vissers, T. Van Goethem, W. Joosen, and N. Nikiforakis, "Maneuvering around clouds: Bypassing cloud-based security providers," in *ACM CCS*, 2015.

[11] A. Yaar, A. Perrig, and D. Song, "SIFF: A stateless Internet flow filter to mitigate DDoS flooding attacks," in *IEEE Symposium on Security and Privacy*, 2004.

[12] X. Yang, D. Wetherall, and T. Anderson, "A DoS-limiting network architecture," in *ACM SIGCOMM*, 2005.

[13] X. Liu, X. Yang, and Y. Xia, "NetFence: Preventing Internet denial of service from inside out," *ACM SIGCOMM CCR*, 2011.

[14] "AT&T denial of service protection," http://soc.att.com/1IllUec, Accessed on 2015.

[15] D. Kim, J. T. Chiang, Y.-C. Hu, A. Perrig, and P. Kumar, "CRAFT: A new secure congestion control architecture," in *ACM CCS*, 2010.

[16] B. Parno, D. Wendlandt, E. Shi, A. Perrig, B. Maggs, and Y.-C. Hu, "Portcullis: Protecting connection setup from denial-of-capability attacks," in *ACM SIGCOMM CCR*, 2007.

[17] P. Mittal, D. Kim, Y.-C. Hu, and M. Caesar, "Mirage: Towards deployable DDoS defense for web applications," *arXiv preprint arXiv:1110.1060*, 2011.

[18] C. Basescu, R. M. Reischuk, P. Szalachowski, A. Perrig, Y. Zhang, H.-C. Hsiao, A. Kubota, and J. Urakawa, "SIBRA: Scalable Internet bandwidth reservation architecture," in *NDSS*, 2016.

[19] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker, "DDoS defense by offense," in *ACM SIGCOMM*, 2006.

[20] S. Peter, U. Javed, Q. Zhang, D. Woos, T. Anderson, and A. Krishnamurthy, "One tunnel is (often) enough," in *ACM SIGCOMM*, 2014.

[21] "Shared Whois Project," https://en.wikipedia.org/wiki/Shared_Whois_Project, Accessed on 2015.

[22] M. Kührer, T. Hupperich, C. Rossow, and T. Holz, "Exit from hell? Reducing the impact of amplification DDoS attacks," in *USENIX Security Symposium*, 2014.

[23] X. Zhang, H.-C. Hsiao, G. Hasker, H. Chan, A. Perrig, and D. G. Andersen, "SCION: Scalability, control, and isolation on next-generation networks," in *IEEE S&P*, 2011.

[24] "CloudFlare," https://www.cloudflare.com, Accessed on 2015.

[25] S. Sundaresan, W. De Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescapè, "Broadband Internet performance: a view from the gateway," in *ACM SIGCOMM*, 2011.

[26] "Whois lookup," https://www.whois.net/, 2016.

[27] T. Herbert, "UDP encapsulation in Linux," in *The Technical Conference on Linux Networking*, 2015.

[28] S. Gueron, "Intel advanced encryption standard (AES) instructions set," *White Paper, Intel*, 2010.

[29] "Helion technology, AES cores," http://www.heliontech.com/aes.htm, 2010.

[30] K. Akdemir, M. Dixon, W. Feghali, P. Fay, V. Gopal, J. Guilford, E. Ozturk, G. Wolrich, and R. Zohar, "Breakthrough AES performance with Intel® AES new instructions," *Intel white paper*, 2010.

[31] "AS relationships – CIDR report," http://www.caida.org/data/as-relationships/, Accessed on 2015.

[32] X. Dimitropoulos, D. Krioukov, M. Fomenkov, B. Huffaker, Y. Hyun, G. Riley et al., "AS relationships: Inference and validation," *ACM SIGCOMM CCR*, 2007.

[33] L. Gao, T. G. Griffin, and J. Rexford, "Inherently safe backup routing with BGP," in *IEEE INFOCOM*, 2001.

[34] S. Goldberg, M. Schapira, P. Hummon, and J. Rexford, "How secure are secure interdomain routing protocols," *ACM SIGCOMM CCR*, 2011.

[35] "AS Names - CIDR Report," http://www.cidr-report.org/as2.0/autnums.html, Accessed on Dec 2015.

[36] "PlanetLab: An open platform for developing, deploying, and accessing planetary-scale services," https://www.planet-lab.org/, Accessed on 2015.

[37] "NS-3: a discrete-event network simulator," http://www.nsnam.org/, Accessed on 2015.

[38] A. Kuzmanovic and E. W. Knightly, "Low-rate TCP-targeted denial of service attacks: the shrew vs. the mice and elephants," in *ACM SIGCOMM*, 2003.

[39] G. Appenzeller, I. Keslassy, and N. McKeown, "Sizing router buffers," in *ACM SIGCOMM*, 2004.

[40] D.-M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Computer Networks and ISDN Systems*, 1989.

[41] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker, "Controlling high bandwidth aggregates in the network," *ACM SIGCOMM*, 2002.

[42] C. Dixon, T. E. Anderson, and A. Krishnamurthy, "Phalanx: Withstanding multimillion-node botnets," in *USENIX NSDI*, 2008.

[43] X. Liu, A. Li, X. Yang, and D. Wetherall, "Passport: Secure and adoptable source authentication," in *USENIX NSDI*, 2008.

[44] S. Savage, D. Wetherall, A. Karlin, and T. Anderson, "Practical network support for IP traceback," *ACM SIGCOMM*, 2000.

[45] D. X. Song and A. Perrig, "Advanced and authenticated marking schemes for IP traceback," in *IEEE INFOCOM*, 2001.

[46] K. J. Argyraki and D. R. Cheriton, "Active Internet traffic filtering: Real-time response to denial-of-service attacks," in *USENIX ATC*, 2005.

[47] J. Ioannidis and S. M. Bellovin, "Implementing pushback: Router-based defense against DDoS attacks," *USENIX NSDI*, 2002.

[48] X. Liu, X. Yang, and Y. Lu, "To filter or to authorize: Network-layer DoS defense against multimillion-node botnets," in *ACM SIGCOMM*, 2008.

[49] A. D. Keromytis, V. Misra, and D. Rubenstein, "SOS: Secure overlay services," *ACM SIGCOMM*, 2002.

[50] D. Naylor et al., "XIA: Architecting a more trustworthy and evolvable Internet," *ACM SIGCOMM*, 2014.

[51] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker, "Accountable internet protocol (AIP)," in *ACM SIGCOMM*, 2008.

[52] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey, "Bohatei: Flexible and elastic DDoS defense," in *USENIX Security Symposium*, 2015.

[53] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson, "FRESCO: Modular composable security services for software-defined detworks," in *NDSS*, 2013.

[54] Y. Gilad, A. Herzberg, M. Sudkovitch, and M. Goberman, "CDN-on-demand: An affordable DDoS defense via untrusted clouds," in *NDSS*, 2016.

[55] F. Chen, R. K. Sitaraman, and M. Torres, "End-user mapping: Next generation request routing for content delivery," in *ACM SIGCOMM*, 2015.

[56] C. Livadas, R. Walsh, D. Lapsley, and W. T. Strayer, "Using machine learning technliques to identify botnet traffic," in *IEEE LCN*, 2006.

[57] A. Karasaridis, B. Rexroad, and D. Hoeflin, "Wide-scale botnet detection and characterization," in *Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets*, 2007.

[58] "Tor Project: Anonymity Online," https://www.torproject.org.

[59] "Tor Metrics," https://metrics.torproject.org.

[60] "Who Uses Tor?" https://www.torproject.org/about/torusers.html.en.

[61] N. Hopper, "Protecting Tor from botnet abuse in the long term," Tech. Rep. 2013-11-001, The Tor Project, Tech. Rep., 2013.

[62] S. Khattak, D. Fifield, S. Afroz, M. Javed, S. Sundaresan, V. Paxson, S. J. Murdoch, and D. McCoy, "Do you see what I see? Differential treatment of anonymous users," in *NDSS*, 2016.

[63] M. Perry, "The trouble with Cloudflare," https://blog.torproject.org/blog/trouble-cloudflare, 2016.

[64] L. Constantin, "Tor network used to command Skynet botnet," http://www.computerworld.com/article/2493980/malware-vulnerabilities/tor-network-used-to-command-skynet-botnet.html, 2012.

[65] J. Dunn, "Mevade botnet miscalculated effect on Tor network, says Damballa," http://www.techworld.com/news/security/mevade-botnet-miscalculated-effect-on-tor-network-says-damballa-3468988/, 2013.

[66] G. Gottesman, "RSA uncovers new POS malware operation stealing payment card & personal information," https://blogs.rsa.com/rsa-uncovers-new-pos-malware-operation-stealing-payment-card-personal-information/, 2014.

[67] "How to handle millions of new Tor clients," https://blog.torproject.org/blog/how-to-handle-millions-new-tor-clients, 2013.

[68] N. Hopper, "Challenges in protecting Tor hidden services from botnet abuse," in *International Conference on Financial Cryptography and Data Security*, 2014.

[69] M. V. Barbera, V. P. Kemerlis, V. Pappas, and A. D. Keromytis, "CellFlood: Attacking Tor onion routers on the cheap," in *ESORICS*, 2013.

[70] N. Borisov, G. Danezis, P. Mittal, and P. Tabriz, "Denial of service or denial of security?" in *ACM CCS*, 2007.

[71] P. Mittal, A. Khurshid, J. Juen, M. Caesar, and N. Borisov, "Stealthy traffic analysis of low-latency anonymous communication using throughput fingerprinting," in *ACM CCS*, 2011.

[72] S. J. Murdoch and G. Danezis, "Low-cost traffic analysis of Tor," in *IEEE S&P*, 2005.

[73] D. Chaum, "Blind signatures for untraceable payments," in *Advances in cryptology*. Springer, 1983.

[74] Y. Sun, A. Edmundson, L. Vanbever, O. Li, J. Rexford, M. Chiang, and P. Mittal, "RAPTOR: Routing attacks on privacy in Tor," in *USENIX Security Symposium*, 2015.

[75] Google, "reCAPTCHA," https://www.google.com/recaptcha/intro/index.html.

[76] D. Goulet and G. Kadianakis, "Random number generation during Tor voting," https://gitweb.torproject.org/torspec.git/tree/proposals/250-commit-reveal-consensus.txt, 2015.

[77] D. Chaum and E. Van Heyst, "Group signatures," in *Workshop on the Theory and Application of of Cryptographic Techniques*, 1991.

[78] L. Chen, "Access with pseudonyms," in *Cryptography: Policy and Algorithms*, 1996.

[79] R. Jansen and A. Johnson, "Safely measuring Tor," in *ACM CCS*, 2016.

[80] M. Abe and T. Okamoto, "Provably secure partially blind signatures," in *Advances in Cryptology CRYPTO*, 2000.

[81] A. Serjantov and G. Danezis, "Towards an information theoretic metric for anonymity," in *Privacy Enhancing Technologies*, 2003.

[82] C. Diaz, S. Seys, J. Claessens, and B. Preneel, "Towards measuring anonymity," in *Privacy Enhancing Technologies*, 2003.

[83] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Computing and Communications Review*, 2001.

[84] "OpenSSL: Cryptography and SSL/TLS toolkit." https://www.openssl.org, Accessed on 2016.

[85] "PyCrypto: The Python cryptography toolkit." https://www.dlitz.net/software/pycrypto/, Accessed on 2016.

[86] "JavaScript library of crypto standards." https://github.com/brix/crypto-js, Accessed on 2016.

[87] "Big integer library." http://evgenus.github.io/bigint-typescript-definitions/index.html#bigint, Accessed on 2016.

[88] R. Jansen and N. Hopper, "Shadow: Running Tor in a box for accurate and efficient experimentation," in *NDSS*, 2012.

[89] Z. Liu, J. Hao, Y.-C. Hu, and M. Bailey, "MiddlePolice: Toward enforcing destination-defined policies in the middle of the Internet," in *ACM CCS*, 2016.

[90] B. Donohue, "How much does a botnet cost?" https://threatpost.com/how-much-does-botnet-cost-022813/77573/, 2013.

[91] M. Motoyama, K. Levchenko, C. Kanich, D. McCoy, G. M. Voelker, and S. Savage, "Re: CAPTCHAs-understanding CAPTCHA-solving services in an economic context." in *USENIX Security Symposium*, 2010.

[92] R. Jansen, "Notes and FAQs of Shadow Simulator," https://github.com/shadow/shadow/wiki/4-Notes-and-FAQs, Accessed on 2016.

[93] R. Dingledine and N. Mathewson, "Tor Path Specification," https://gitweb.torproject.org/torspec.git/tree/path-spec.txt, 2016.

[94] "Tor Atlas: a web application to learn about currently running Tor relays," https://atlas.torproject.org, Accessed on 2016.

[95] I. Goldberg, D. Stebila, and B. Ustaoglu, "Anonymity and one-way authentication in key exchange protocols," *Designs, Codes and Cryptography*, 2013.

[96] R. Henry and I. Goldberg, "Formalizing anonymous blacklisting systems," in *IEEE S&P*, 2011.

[97] I. B. Damgård, "Payment systems and credential mechanisms with provable security against abuse by individuals," in *Conference on the Theory and Application of Cryptography*, 1988.

[98] S. G. Stubblebine, P. F. Syverson, and D. M. Goldschlag, "Unlinkable serial transactions: Protocols and applications," *ACM Transactions on Information and System Security (TISSEC)*, 1999.

[99] A. Lysyanskaya, R. L. Rivest, A. Sahai, and S. Wolf, "Pseudonym systems," in *International Workshop on Selected Areas in Cryptography*, 1999.

[100] P. C. Johnson, A. Kapadia, P. P. Tsang, and S. W. Smith, "Nymble: Anonymous IP-address blocking," in *Privacy Enhancing Technologies*, 2007.

[101] P. P. Tsang, A. Kapadia, C. Cornelius, and S. W. Smith, "Nymble: Blocking misbehaving users in anonymizing networks," *IEEE Transactions on Dependable and Secure Computing*, 2011.

[102] R. Henry and I. Goldberg, "Extending Nymble-like systems," in *IEEE S&P*, 2011.

[103] Z. Lin and N. Hopper, "Jack: Scalable accumulator-based Nymble system," in *ACM Workshop on Privacy in the Electronic Society*, 2010.

[104] P. P. Tsang, M. H. Au, A. Kapadia, and S. W. Smith, "PEREA: Towards practical TTP-free revocation in anonymous authentication," in *ACM CCS*, 2008.

[105] P. P. Tsang, M. H. Au, A. Kapadia, and S. W. Smith, "BLAC: Revoking repeatedly misbehaving anonymous users without relying on TTPs," *ACM Transactions on Information and System Security (TISSEC)*, 2010.

[106] P. Lofgren and N. Hopper, "FAUST: Efficient, TTP-free abuse prevention by anonymous whitelisting," in *ACM Workshop on Privacy in the Electronic Society*, 2011.

[107] E. Brickell and J. Li, "Enhanced privacy ID: A direct anonymous attestation scheme with enhanced revocation capabilities," in *Proceedings of the 2007 ACM Workshop on Privacy in Electronic Society*, 2007.

[108] R. Dingledine, D. S. Wallach et al., "Building incentives into Tor," in *Financial Cryptography and Data Security*, 2010.

[109] R. Jansen, N. Hopper, and Y. Kim, "Recruiting new Tor relays with BRAIDS," in *ACM CCS*, 2010.

[110] R. Jansen, A. Johnson, and P. Syverson, "LIRA: Lightweight incentivized routing for anonymity," in *NDSS*, 2013.

[111] M. Ghosh, M. Richardson, B. Ford, and R. Jansen, "A TorPath to TorCoin: Proof-of-bandwidth altcoins for compensating relays," in *Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs)*, 2014.

[112] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *ACM SIGCOMM*, 2011.

[113] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *ACM SIGCOMM*, 2015.

[114] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *ACM IMC*, 2009.

[115] T. Benson, A. Akella, and D. Maltz, "Network traffic characteristics of data centers in the wild," in *ACM IMC*, 2010.

[116] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos, "ElasticSwitch: Practical work-conserving bandwidth guarantees for cloud computing," in *ACM SIGCOMM*, 2013.

[117] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, "SecondNet: A data center network virtualization architecture with bandwidth guarantees," in *ACM CoNext*, 2010.

[118] A. Shieh, S. Kandula, A. G. Greenberg, C. Kim, and B. Saha, "Sharing the data center network." in *USENIX NSDI*, 2011.

[119] T. Lam, S. Radhakrishnan, A. Vahdat, and G. Varghese, *NetShare: Virtualizing Data Center Networks Across Services*. [Department of Computer Science and Engineering], University of California, San Diego, 2010.

[120] S. Hu, W. Bai, K. Chen, C. Tian, Y. Zhang, and H. Wu, "Providing bandwidth guarantees, work conservation and low latency simultaneously in the cloud," in *IEEE INFOCOM*, 2016.

[121] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma, "Application-driven bandwidth guarantees in datacenters," in *ACM SIGCOMM*, 2014.

[122] J. Mudigonda, P. Yalagandula, J. Mogul, B. Stiekes, and Y. Pouffary, "NetLord: A scalable multi-tenant network architecture for virtualized datacenters," in *ACM SIGCOMM CCR*, 2011.

[123] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive, "A flexible model for resource management in virtual private networks," in *ACM SIGCOMM CCR*, 1999.

[124] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "FairCloud: Sharing the network in cloud computing," in *ACM SIGCOMM*, 2012.

[125] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg, "EyeQ: Practical network performance isolation at the edge," *USENIX REM*, 2013.

[126] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes, "Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks." in *WIOV*, 2011.

[127] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM*, 2008.

[128] J. Crowcroft and P. Oechslin, "Differentiated end-to-end Internet services using a weighted proportional fair sharing TCP," *ACM SIGCOMM CCR*, 1998.

[129] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska, "End-to-end performance isolation through virtual datacenters," in *USENIX OSDI*, 2014.

[130] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: A scalable and flexible data center network," in *ACM SIGCOMM*, 2009.

[131] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano et al., "Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network," in *ACM SIGCOMM*, 2015.

[132] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking data centers randomly." in *USENIX NSDI*, 2012.

[133] J.-Y. Shin, B. Wong, and E. G. Sirer, "Small-world datacenters," in *ACM SoCC*, 2011.

[134] D. Eppstein, "Finding the k shortest paths," in *IEEE FOCS*, 1994.

[135] P. Bodík, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica, "Surviving failures in bandwidth-constrained datacenters," in *ACM SIGCOMM*, 2012.

[136] Q. Xu, C. Lumezanu, Z. Liu, N. Arora, A. Sharma, H. Zhang, and G. Jiang, "Optimization framework for multi-tenant data centers," Patent U.S. App. 14/830,303, 2015.

[137] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some simplified NP-complete problems," in *ACM STOC*, 1974.

[138] "Configuring the DSCP-to-DSCP-mutation map for Cisco switches," http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst2960/software/release/12-2_55_se/configuration/guide/scg_2960/swqos.html#wp1028614.

[139] "The netfilter.org project," http://www.netfilter.org.

[140] "Netlink: Communication between kernel and user space," http://man7.org/linux/man-pages/man7/netlink.7.html.

[141] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese et al., "CONGA: Distributed congestion-aware load balancing for datacenters," in *ACM SIGCOMM*, 2014.

[142] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter TCP (D2TCP)," *ACM SIGCOMM CCR*, 2012.

[143] C.-Y. Hong, M. Caesar, and P. Godfrey, "Finishing flows quickly with preemptive scheduling," *ACM SIGCOMM CCR*, 2012.

[144] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pFabric: Minimal near-optimal datacenter transport," *ACM SIGCOMM CCR*, 2013.

[145] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic flow scheduling for commodity data centers," in *USENIX NSDI*, 2015.

[146] "The OpenDayLight project," https://www.opendaylight.org.

[147] D. Xie, N. Ding, Y. C. Hu, and R. Kompella, "The only constant is change: Incorporating time-varying network reservations in data centers," *ACM SIGCOMM CCR*, 2012.

[148] G. Kumar, S. Kandula, P. Bodik, and I. Menache, "Virtualizing traffic shapers for practical resource allocation," in *Workshop on HotCloud*, 2013.

[149] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese et al., "P4: Programming protocol-independent packet processors," *ACM SIGCOMM CCR*, 2014.

[150] "Amazon data center size," http://huanliu.wordpress.com/2012/03/13/amazon-data-center-size/.

[151] "Amazon EC2 grows 62% in 2 years," http://huanliu.wordpress.com/2014/02/26/amazon-ec2-grows-62-in-2-years/.

[152] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram et al., "Network virtualization in multi-tenant datacenters," in *USENIX NSDI*, 2014.

[153] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM*, 2008.

[154] "OpenFlow," https://www.opennetworking.org/sdn-resources/onf-specifications/openflow.

[155] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "VXLAN: A framework for overlaying virtualized layer 2 networks over layer 3 networks," *draftmahalingam-dutt-dcops-vxlan-01. txt*, 2012.

[156] M. Sridharan, K. Duda, I. Ganga, A. Greenberg, G. Lin, M. Pearson, P. Thaler, C. Tumuluri, N. Venkataramiah, and Y. Wang, "NVGRE: Network virtualization using generic routing encapsulation," *IETF draft*, 2011.

[157] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks." in *USENIX NSDI*, 2010.

[158] S. Shin and G. Gu, "CloudWatcher: Network security monitoring using OpenFlow in dynamic cloud network," in *IEEE ICNP*, 2012.

[159] S. Shin, H. Wang, and G. Gu, "A first step toward network security virtualization: From concept to prototype," *IEEE TIFS*, 2015.

[160] W. J. Dally and B. P. Towles, *Principles and Practices of Interconnection Networks*. Elsevier, 2004.

[161] P. J. Frantz and G. O. Thompson, "VLAN frame format," Patent U.S. 5,959,990, 1999.

[162] R. Enns, M. Bjorklund, and J. Schoenwaelder, "NETCONF configuration protocol," *Network*, 2011.

[163] "Trema: Full-stack OpenFlow framework in Ruby and C," http://trema.github.io/trema/.

[164] "Stacked VLAN Processing," http://www.cisco.com/c/en/us/td/docs/ios/12_0s/feature/guide/qinq.html.

[165] D. Levin, M. Canini, S. Schmid, F. Schaffert, A. Feldmann et al., "Panopticon: Reaping the benefits of incremental SDN deployment in enterprise networks," in *USENIX ATC*, 2014.

[166] N. Dukkipati and N. McKeown, "Why flow-completion time is the right metric for congestion control," *ACM SIGCOMM*, 2006.

[167] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined WAN," *ACM SIGCOMM*, 2013.

[168] H. Lu, N. Arora, H. Zhang, C. Lumezanu, J. Rhee, and G. Jiang, "HybNET: Network manager for a hybrid network infrastructure," in *Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference*, 2013.

[169] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. M. Parulkar, "Can the production network be the testbed?" in *USENIX OSDI*, 2010.

[170] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," in *ACM CoNext*, 2011.

[171] S. Kandula, D. Katabi, S. Sinha, and A. Berger, "Dynamic load balancing without packet reordering," *ACM SIGCOMM CCR*, 2007.

[172] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "DeTail: Reducing the flow completion time tail in datacenter networks," *ACM SIGCOMM*, 2012.

[173] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: A scalable fault-tolerant layer 2 data center network fabric," in *ACM SIGCOMM*, 2009.

[174] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *ACM SIGCOMM*, 2013.

[175] "The CAIDA DNS root/gTLD RTT dataset," http://www.caida.org/data/passive/dns_root_gtld_rtt_dataset.xml, Accessed on 2016.

[176] "TC: A Tor control protocol (version 1)." https://gitweb.torproject.org/torspec.git/tree/control-spec.txt.

[177] D. Johnson, "Stem is a Python controller library for Tor," https://stem.torproject.org.